Refining Expression DAGs in Exact-Decisions Number Types

Dissertation zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.), angenommen durch die Fakultät für Informatik der Otto-von-Guericke Universität Magdeburg von

Martin Wilhelm
geb. am 10.01.1991 in Suhl

Gutachter

Prof. Dr. Stefan Schirra
Dr. Joris van der Hoeven
Dr. Monique Teillaud

OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

Magdeburg, den 4. Februar 2020

Zusammenfassung

Exakte Zahlentypen spielen eine zentrale Rolle bei der Entwicklung von robusten Algorithmen in dem Gebiet der Algorithmischen Geometrie, in welchem kombinatorische Entscheidungen auf der Grundlage von numerischen Berechnungen erfolgen. Das „Exact Computation Paradigm“ besagt, dass Robustheit erlangt werden kann, indem sichergestellt wird, dass alle Entscheidungen, die während der Ausführung eines Algorithmus getroffen werden, korrekt sind. In einem Exakte-Entscheidungen-Zahlentyp wird die Berechnung eines Programms in einem gerichteten azyklischen Graphen, einem so genannten arithmetischen Ausdrucksgraphen, abgespeichert. Während eines Entscheidungsprozesses werden alle Operationen in dem Ausdrucksgraphen so lange mit sich erhöhender Genauigkeit ausgewertet, bis eine exakte Entscheidung garantiert werden kann.

In der vorliegenden Arbeit wird die Auswertung eines Ausdrucksgraphen für Berechnungen mit großen Graphen oder großer Genauigkeit verbessert. Ausdrucksgraphen mit schlechter Struktur können einen quadratischen Anstieg in der Laufzeit zur Folge haben, wenn die Berechnungen groß werden. Diese Effekte können durch Methoden zum Balancieren von sowohl der verwendeten Fehlerverteilung als auch der Struktur des Graphen abgemildert werden. In beiden Fällen werden optimale Strategien vorgestellt, bewiesen und experimentell ausgewertet. Das Balancieren von Fehlerschranken ist sehr vielseitig einsetzbar und kann einen großen Teil der Kosten, die mit der unbalancierten Struktur verknüpft sind, verringern. Das Anwendungsfeld für die Methode der Neustrukturierung ist eingeschränkter. Wenn sie eingesetzt werden kann, führt dies jedoch häufig zu starken Verbesserungen der Laufzeit. Sowohl Situationen, in denen Neustrukturieren vorteilhaft ist, als auch Situationen, in denen Neustrukturierung nachteilig ist, werden in der Arbeit beschrieben. Sobald eine hohe Genauigkeit erforderlich ist, kann die Laufzeit durch die Parallelisierung der arithmetischen Operationen in dem Ausdrucksgraphen verringert werden. Methoden für eine effektive Parallelisierung werden beschrieben, analysiert und experimentell ausgewertet. Eine Neustrukturierung kann dazu benutzt werden, die Parallelisierbarkeit eines Ausdrucksgraphens zu erhöhen. Beide Balanciermethoden werden im Hinblick auf eine parallele Umgebung ausgewertet.

Abstract

Exact number types play a central role in the development of robust algorithms in the field of computational geometry, where combinatorial decisions are made on the basis of numerical computations. The Exact Computation Paradigm states that for achieving robustness it is sufficient to guarantee that all decisions made during the execution of an algorithm are correct. In an exact-decisions number type, the computation history of a program is stored in a directed acyclic graph, a so-called arithmetic expression dag. During a decision process, all operations in the expression dag are evaluated with increasing accuracy until an exact decision can be guaranteed.

In this work, the evaluation process of an expression dag is improved for computations with large expression dags or high accuracy. Badly structured expression dags can lead to a quadratic increase in the evaluation time if computations get large. These effects are mitigated by balancing methods for both the error distribution appearing during an evaluation and the graph structure itself. In both cases, optimal strategies are proposed, proven and experimentally evaluated. Error bound balancing is very versatile and can reduce a large amount of the cost associated with unbalanced structures. The field of application for restructuring is more narrow, but when it can be used it often leads to strong improvements on the running time. Both situations in which restructuring is beneficial and situations in which restructuring can be detrimental to the running time are described in this work. If a high accuracy is required, the evaluation time can be reduced by a parallelization of the arithmetic operations in the expression dag. Methods for an effective parallelization are described, analyzed and experimentally evaluated. Restructuring can be employed to increase the parallelizability of an expression dag. Both balancing methods are evaluated with respect to a parallel environment.

Contents

1 Introduction

The importance of information technology in the modern society leads to an ever-rising demand on efficient algorithms. Many new algorithms are proposed each year by computer science researchers in order to catch up to this demand. Alas, a rocky road lies between the development of a theoretical algorithm and its final application in practice. The theoretical focus on asymptotic worst-case analysis for abstract machines leads to gaps between theory and practice. Sometimes, theoretically optimal algorithms are outperformed by algorithms deemed inefficient when confronted with real-world data. In other cases, correct implementations cause undefined behavior when assumptions made for the abstract machine model do not transfer to real processors. In the last twenty years, the field of algorithm engineering emerged as an increasing number of researchers developed techniques to transfer theoretical results into practice. Algorithm engineering aims to define standard methodologies and realistic assumptions for the analysis of algorithms and to develop practically efficient algorithms by following a strict empirical design cycle [DFI03; San09]. This cycle encompasses the entire design process, including the modeling of both the problem and the target machine, the design and analysis of algorithms, the actual implementation of the algorithms and the design and implementation of meaningful experiments. Relevant aspects for each design step are the time and space used by the final program, but also the simplicity, scalability and, naturally, the correctness of the implementation [MS10]. An important aspect for achieving correctness is the robustness of an algorithm with respect to computation errors, i.e., with respect to deviations of the actual computed values to the values predicted by the machine model. A common cause for such deviations is the implementation of operations on real numbers through floating-point arithmetic in algorithms that are based on the so-called real RAM model.

1.1 The Real RAM Model

Most of modern algorithm designs are based on the *random access machine* (RAM) model introduced in 1973 by Stephen Cook and Robert Reckhow [CR73]. The model allows any integer number to be arbitrarily stored and read in time linear to the size of its binary representation without any additional access cost. Likewise, additions of two integers can be performed in time linear to the sum of their representation sizes. The RAM model exists in many different variations. Notably, with the unit cost RAM and the word RAM, machine models were introduced that allow basic access operations as well as basic arithmetic operations on integers at unit cost [AHU74; FW93b]. While reasonably powerful for the analysis of discrete algorithms, the applicability of these

RAM

RAM models is limited when used in a continuous environment. The lack of operations on real numbers makes it hard to derive meaningful asymptotic bounds or even to prove the correctness of the respective algorithms.

Non-integral operations, such as divisions or square roots, occur frequently if algorithms are designed to work on lines, circles or other geometric objects. Geometric algorithms play a fundamental role in many fields of computer science. The study of geometric problems from a computational point of view, including the design and analysis of geometric algorithms, is subsumed under the term *computational geometry.* Applications of computational geometry range from fundamental problems in computer graphics, robotics and geographic information systems to use cases in statistics, biology and physics [GO04]. In order to cope with the special requirements of a continuous setting, Michael Shamos extended the RAM model of computation by allowing the representation of real numbers as well as an exact computation of all analytical functions at unit cost.

real RAM

The resulting machine model is called the *real RAM* model of computation [Sha78]. To date, virtually all geometric algorithms rely on the real RAM for both their correctness and their complexity analysis.

1.2 The Robustness Problem in Computational Geometry

The real RAM is an exceptionally powerful machine model. In actual computers, real numbers cannot be represented exactly. They are therefore usually approximated by floating-point numbers with a fixed size. Consequently, operations on these numbers introduce errors to the computation. For a reliable implementation, it is crucial that such errors do not substantially change the expected output. This property is expressed

robust

stable

by the principles of *robustness* and *stability.* An algorithm is called *robust* if its result is the correct result for some perturbation of the input. An algorithm is called *stable* if moreover said perturbation is small with respect to an appropriate metric [For89]. While not necessarily optimal, a stable algorithm is guaranteed to always produce meaningful results, despite the presence of rounding errors.

Unfortunately, geometric algorithms based on the real RAM model are not guaranteed to be stable. To make things worse, they are not even guaranteed to be robust. Algorithms may fail, produce wrong output or no output at all if confronted with rounding errors [Sch00]. The reason for this undesirable outcome lies in the interdependence between numerical and combinatorial computations that is present in most geometrical settings. Although a wide variety of numerical methods exist to control the maximum error induced by the computation, these methods by themselves do not help in the context of combinatorial decisions. When confronted with combinatorial algorithms, even the slightest deviation from the correct result of any computation during the algorithm may lead to decisions that are wrong or, in the worst case, contradict previously established properties. There is a variety of examples for strikingly wrong output created as a consequence of inconsistencies caused by rounding errors, including algorithms for convex hulls, Delaunay triangulations and boundary representations of solid objects [She97; Hof01; Ket+08; Mör15b]. An example for a failed iterative construction of a convex hull

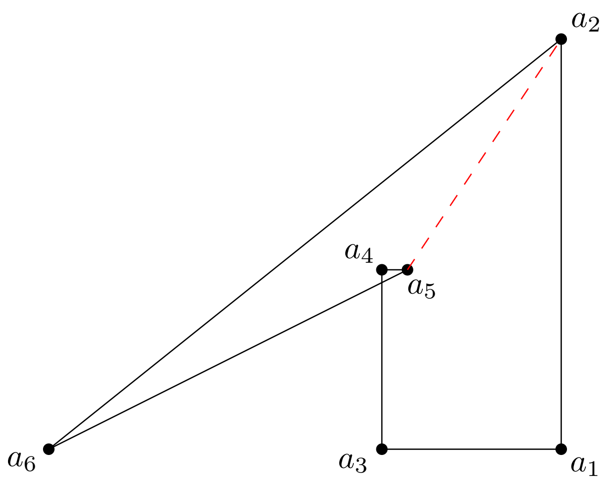


Figure 1.1: The output of an iterative convex hull algorithm that made one wrong decision as described by Kettner et al. [Ket+08]. During the algorithm, the points $a_1, \ldots, a_6$ are added iteratively to the convex hull. If a point lies outside, the algorithm removes a visible edge and all neighboring visible edges. By falsely deciding that a_5 lies left of $\overline{a_4 a_2}$, the property that for each point all visible edges form a polygonal chain is violated. When inserting a_6, the algorithm removes $\overline{a_5 a_2}$ and falsely decides that there are no further visible edges.

is shown in Figure 1.1.

Designing algorithms that cannot lead to inconsistencies is difficult. Steven Fortune introduces the notion of a *parsimonious algorithm*, which exclusively performs tests if their outcome cannot be derived from previous tests [For89]. While such algorithms are not necessarily robust, they provide a weak form of robustness, where a result is guaranteed to be numerically close to the correct result for the given input [Knu92]. Only few examples for parsimonious algorithms can be found in literature [Sch91; Knu92; ZFB93]. A technique called the *topological-oriented approach* achieves parsimoniousness by altering the input to fit the decisions made by the algorithm [SI89; Sug+00]. In general, determining whether a statement is a logical consequence of the previous statements is as hard as the existential theory of reals [Mnë88; For89], which, in turn, lies in PSPACE and is shown to be NP-hard [Can88]. Consequently, several techniques have been developed to obtain robust or even stable algorithms by capitalizing on characteristics of floating-point arithmetic, thereby avoiding the necessity of parsimoniousness. They usually involve a perturbation of the input (e.g. [Mil88; For89; GSS89; HHK89; FB91]) or a relaxation of the conditions imposed on the used geometrical objects (e.g. [Mil88; FM91; Che+01]). Several geometric problems are shown to be robustly solvable in reasonable times by applying such techniques [Sch00; CWK02].

Robust algorithms are hard to design and differ significantly from their counterparts in the real RAM model. Designing robust algorithms with respect to floating-point numbers effectively equates to designing algorithms for the standard RAM model, thereby negating all benefits introduced through the real RAM. By shifting the focus away from the robust design of algorithms to the design of exact number types, the real RAM model can be retained without giving up on robustness [Yap97]. If the underlying number types are exact, algorithms that are designed for the real RAM maintain their correctness when they are implemented on a real machine. Exact number types have been established in several geometric libraries [MN89; MRS10; Yap98; Yu+10]. Although extremely powerful, their generality comes at a high cost. In practice, using exact computation can make computations arbitrarily slow, especially if the complexity of the computations increases with the problem size [Liu+16]. In a slightly less general approach, instead of basic arithmetic operations, exact predicates, such as an orientation test and an incircle test, are implemented exactly. Implementing these predicates directly is potentially much faster than the usage of a general-purpose number type. Therefore, several authors have focussed on building libraries for these predicates [She97; Lév16; QYZ19].

In this work, we follow the more general approach. Predicate libraries reach their limit if computations get more complex. Nevertheless, in their current state, exact number types are designed primarily for small expressions and may behave suboptimally if expressions get large. This can be the case as soon as objects are the result of several iterations, such as in the computation of minimum link paths in a polygon [KS99]. Aside from computational geometry, efficient handling of large expressions may open up new applications in automated reasoning or even symbolic computation [DMM05; Bod+03]. In the next chapter, we review basic concepts of multiple-precision and exact number types. Chapter 3 introduces a new extension for the exact number type `Real_algebraic`, which allows us to easily exchange and combine various parts of the implementation. The extension forms the basis for experiments presented in the subsequent chapters. In Chapter 4, we explore two strategies for increasing the evaluation performance of large expressions. Both strategies aim for a reduction of the total amount of precision needed during the computation. The first strategy, based on work by Joris van der Hoeven, balances error bounds used during the evaluation of the underlying graph structure. In the second strategy, the graph is instead restructured, mostly based on work by Richard Brent. Chapter 5 both introduces a parallel evaluation strategy for exact number types and examines the behavior of restructuring and error bound balancing on the parallelizability of the evaluation. Finally, in Chapter 6, we compare the impact of various error representations on the evaluation time.

2 Concepts

Every programming language contains primitive integer and floating-point number types. These number types have a small fixed size, depending on the programming language and the processor architecture, and operations with these number types are usually highly optimized through a direct representation in hardware. While very fast, primitive types have a bounded capacity and operations on these types can lead to overflow, underflow or rounding errors. In order to accurately simulate a real RAM, computations with these primitives must be replaced by more elaborate number types, which, to some degree, allow for "exact" computations. In this chapter, we review the concepts of multiple-precision and exact number types. Furthermore, we briefly examine the related concept of effective numbers and describe the differences between the methods developed in the field of computational geometry and the methods developed in the field of computable analysis.

2.1 Multiple-Precision Number Types

The concept of multiple-precision number types builds on, and extends the concept of fixed-precision primitives. A *multiple-precision number type* is a number type that stores values in a format with arbitrary finite size and provides basic arithmetic operations on these values. In contrast to a primitive type, its size is not defined by the system, but can be chosen and adjusted by the program. A *multiple-precision integer type (bigint)* stores integer values in a binary format and supports negation, addition, subtraction, multiplication and integer division. Since these operations always produce integers, they can in principle be exactly represented by a bigint. For operations on large numbers, however, the number type might overflow. We call the maximum number of bits in the binary representation of a representable number the *size* of a bigint.

multiple-precision number type

bigint

A *multiple-precision floating-point type (bigfloat)* represents floating-point values by storing a mantissa m in a bigint and an exponent e in a large integer primitive. Note that storing the exponent in a primitive type sets a theoretical limit to the range of numbers a bigfloat can represent. Depending on its usage, it may be argued that numbers exceeding the range defined by a primitive exponent cannot be created in any realistic scenario. Otherwise, a bigint can be used for the representation of the exponent as well. In this work, we assume the former definition. Bigfloat number types support all basic arithmetic operations as well as roots to an arbitrary degree $d \in \mathbb{N}$. Operations on bigfloats simulate operations over the real numbers. Since only few real numbers can actually be represented by a finite floating-point representation, operations usually return *approximations* to the actual result. The quality of these approximations and, more generally, of the available operations is described by the terms *accuracy* and *precision*.

bigfloat

2.1.1 Accuracy and Precision

Fundamentally, a series of approximations to a value is said to be *accurate* if its maximum deviation from the approximated value is small and *precise* if its variance is small. Before describing these concepts in more detail, we clarify the usage of the terms *size* and *magnitude* in this work.

size, magnitude

Definition 2.1. Let $r \in \mathbb{R}$ be a real number. Then we call the absolute value $|r|$ the *size* of r and we call the binary logarithm of its size, $\log(|r|)$ the *magnitude* of r.

Remark 2.2. In this work, we use the term *absolute* to contrast the meaning of *relative* in various situations. To refer to the absolute value norm of a number, we use the term *size* instead. This definition is not to be confused with the size of a number type, which relates to the number of bits needed to save the data.

accuracy
error

In numerical analysis, the *accuracy of an approximation* describes how close the approximation is to the actual value. The *error of an approximation* describes the deviation of the approximation to the approximated value. Errors can be represented as absolute differences or relative to the size of the approximated value or the approximation. For our purpose, it is more convenient to define a relative error relative to the size of the approximation.

absolute error
relative error

Definition 2.3. Let $z \in \mathbb{R}$ be a number and let $\tilde{z}$ be an approximation to z. Then $\varepsilon = |\tilde{z} - z|$ is called the *(absolute) error* of the approximation. For $\tilde{z} \neq 0$ the *relative error* of the approximation is defined as $\xi = \frac{\varepsilon}{|\tilde{z}|}$.

absolute accuracy
relative accuracy

We define the *(absolute) accuracy* and *relative accuracy* of an approximation to refer to its absolute and relative error.

Remark 2.4. Note that, while accuracy and error are mathematically described by the same value, the concept of accuracy is reciprocal to the concept of error. So when the accuracy is increased, the value representing the accuracy is decreased and vice versa. Likewise, the term *high accuracy* is used to refer to a small error value while *low accuracy* refers to a large error value.

precision

correctly rounded

The *precision* of a bigint is defined as its size, i.e., the maximum number of bits used for a representation. The *precision* of a bigfloat is the size of its mantissa. We say that an operation on a bigfloat is *correctly rounded* with respect to a given rounding mode, if it always returns the correct result or the nearest representable number according to the rounding direction. If a bigfloat operation is correctly rounded, the precision of the number type yields an upper bound to the relative error of the result.

2.1.2 Operation Complexity

The computation time for basic arithmetic operations on primitive integer or floating-point number types is a small constant, which is mostly dependent on the hardware implementation. Nevertheless, there are measurable differences between different operations. Table 2.1 shows running times of all basic operations and root operations for the floating-point primitive `double` in C++ (for a description of the test environment see Section 4.4.1). Negations are very fast. Additions, subtractions and multiplications take nearly the same running time. Divisions, however, are significantly slower, even slower than square root operations. The root of degree three is not natively implemented. Simulating it with help of the library function `pow` makes it much slower than the square root.

Operation	neg	add	sub	mul	div	sqrt	root3
Time (seconds)	0.08	0.50	0.49	0.54	1.38	1.05	15.69

Table 2.1: Running times of the execution of various operations on random `double` numbers in C++. Each operation is executed $3 \cdot 10^8$ times. The results of the operations are assigned, added and multiplied to other floating-point numbers to avoid pipelining effects and the measured time is adjusted by a baseline. The root of degree 3 is determined by computing the 1/3-th power.

Multiple-precision number types must rely on software implementations for the basic operations. On both bigints and bigfloats, comparisons can be done in linear time with respect to the size of the number type. The running time of an arithmetic operation depends on the precision of the number type in which the result should be stored. We call this precision the *precision of the operation*. In particular, if implemented carefully, the running time is not negatively influenced by the precision of the operands, except for catastrophic cancellation in the case of additions. Negations take constant time. Additions and subtractions with precision p can be performed in linear time on integers by a bitwise addition. For floating-point numbers a linear time can be achieved as well by an addition of the mantissas after shifting the mantissa of the operand that is smaller in size by the exponent difference. Note that for large differences in the exponents, the number of bitwise additions can be reduced or a bitwise addition might not be necessary at all. If the numbers are equal or nearly equal in size and of opposite sign, the running time depends on the precision of the operands. Obviously, the asymptotic running time for additions is optimal.

Remark 2.5. In an actual implementation, operations are not performed on single bits but on primitives, which are fused together into a multiple-precision number type. Nevertheless, the general principles transfer naturally from a bitwise analysis.

The multiplication of two bigfloat numbers can be performed by multiplying their mantissas and adding their exponents. Since obviously integer multiplication reduces to floating-point multiplication, their running time is asymptotically equal. Classical long ("schoolbook") multiplication takes quadratic time. Karatsuba was the first to show that multiplication could be done in subquadratic time [KO62]. His divide and conquer approach was later shown to be a special case of a more general algorithm invented by Toom [Too63] and improved by Cook [Coo66]. Schönhage and Strassen introduced the first quasilinear algorithm based on fast fourier transformation, running in time

$\Theta(n \log n \log\log n)$ [SS71]. The optimal asymptotic running time of multiplication is not known by today. While $\Omega(n \log n)$ is conjectured to be a lower bound, non-trivial lower bounds are only known for a special kind of Turing machines [CA69; Che74; PFM74]. The algorithm of Schönhage and Strassen misses the conjectured lower bound by a factor of $\log\log n$. Many years later, Fürer was able to show that this factor can be further reduced [Für09]. Recently, Harvey and van der Hoeven claimed to have found a way to eliminate the additional factor, achieving the conjectured optimal running time of $O(n \log n)$ [HH19]. An overview of the asymptotic running times of these multiplication algorithms is presented in Table 2.2.

Long Multiplication $\Theta(n^2)$	Karatsuba $\Theta(n^{\log 3})$	Toom-Cook $\Theta(n^{1+\varepsilon})$
Schönhage-Strassen $\Theta(n \log n \log\log n)$	Fürer $n \log n \, 2^{O(\log^*(n))}$	Harvey-van der Hoeven $\Theta(n \log n)$

Table 2.2: Asymptotic running times of various multiplication algorithms.

Having an efficient multiplication algorithm is central for an efficient multiple-precision number type since both division and d-th roots can be reduced to multiplication. Many different approaches exist for division algorithms [OF97]. By computing the reciprocal of the divisor with the Newton method (also referred to as Newton-Raphson method [Deu12]), a division can be performed in time $\Theta(M(n))$, where $M(n)$ is the time needed for multiplication [Fly70]. Burnikel and Ziegler introduced a divide-and-conquer division algorithm which runs in $O(M(n) \log n)$. It produces an overhead of $\log n$ for the known efficient multiplication algorithms, but is very fast in practice [BZ98; Has97]. The square root of a number can be reduced to multiplication with Newton's method as well, achieving $O(M(n))$ running time [Alt79]. Using the same approach, roots of arbitrary degree d can be computed in time $O(M(n) \log d)$.

Operation	neg	add	sub	mul	div	sqrt	root3
Time (seconds)	0.01	0.03	0.03	2.71	5.78	3.87	8.18

Table 2.3: Running times of the execution of various operations on random numbers between 1 and 2 of type `mpfr_t` with precision 10^5. Each operation is executed 10^4 times.

In Table 2.3 the running times for the basic operations as well as for square and cubic roots are shown for an efficient bigfloat number type. In comparison to Table 2.1, there is a larger gap between the running times for negation, addition and subtraction and the running times of the remaining operations. In particular, additions and multiplications differ significantly in running time. Note that the running time for negations includes the running time for an assignment operation and therefore is non-zero with respect to the displayed precision.

2.1.3 Multiple-Precision Libraries

There is a wide variety of software implementations for multiple-precision number types. Most major programming languages provide bigint types in their standard libraries, including compiled languages such as Go, C# and Java [Go19; CS18; Jav19], as well as interpreted languages such as JavaScript, PHP and Perl [JS19; PHP19; Prl19]. In Python and Ruby, all integer types are of arbitrary precision by default [Pyt19; Rub19]. A native support of bigfloat types is less prevalent. In many languages they are added through third-party libraries instead. The standard libraries of C and C++ do not support multiple-precision arithmetic. A large number of third-party packages for both integer and floating-point types is available. Multiple-precision integers are heavily used in cryptography. Consequently, many popular cryptographic libraries, such as `Libgcrypt`, `LibTomMath`, `mbed TLS` and `OpenSSL`, provide native bigint implementations [GPG19; LTM19; Mbed19; OSSL19]. Multiple-precision floating-point types are mostly found in geometric libraries. Both the `CGAL` library and the `GeometricTools` library provide implementations of multiple-precision integers, rationals and floats, but with the latter limited to additions and multiplications [CGAL19; Dav19]. On top of integers and rationals, the `LEDA` library as well as the `CORE` library provide bigfloat data types supporting all basic arithmetic operations and roots of arbitrary degree [MN99; Yu+10]. Beside these libraries, there are various stand-alone bigfloat implementations. However, few of these implementations appear to be sufficiently curated. Better maintained projects include the `boost` multiprecision library, the Class Library for Numbers (`CLN`), the `bigz` library and the `LibBF` library [MK19; HK14; SVH89; Bel19].

Perhaps the most notable stand-alone library providing multiple-precision arithmetic is the GNU Multiple Precision Arithmetic Library (`GMP`) [GMP16]. The library offers highly optimized operations on multiple-precision integer, rational and floating-point types. For multiplication, `GMP` uses a variety of different algorithms depending on the size of the operands and the required precision. Starting with long multiplication for small precisions, it utilizes Karatsuba's algorithm and several modified versions of the Toom-Cook algorithm up to the Schönhage-Strassen multiplication. Likewise, various division algorithms are employed, depending on the operand sizes. Beside long division, the Burnikel-Ziegler division and a variant of Newton's method based on Barret reduction ([Bar86]) are used for large operands. In addition, several special cases are identified in which even faster algorithms can be used, such as cases in which the result or one of the operands is very small or cases in which the result can be represented exactly. Root operations of any degree are reduced to multiplication by (variants of) Newton's method.

Several other implementations of bigint and bigfloat arithmetic exist that are based on `GMP`. The `MPIR` library provides optimized multiple-precision number types with a special focus on parallelizability based on `GMP` [Gla+17]. The `MPFR` library provides an adaptation of the bigfloat implementation of `GMP`, which satisfies most properties demanded by the IEEE 754 floating-point standard [Fou+07]. Among other improvements, this includes correct rounding of the results of an expression with respect to any of four freely selectable rounding modes [IE754]. Due to its high efficiency and its inherent error guarantees, `MPFR` is well-suited as a basis for the development of exact number types.

2.2 Exact Number Types

Multiple-precision number types make it possible to arbitrarily increase the precision at which a result is computed. If numerical inaccuracies are expected that may lead to problems, a higher precision for the number type can be chosen to avoid them. This approach is called the *Fixed Precision Paradigm.* For solving the robustness problem in computational geometry, increasing the precision of the underlying number types is by itself not sufficient (cf. Section 1.2). Operations like divisions and roots lead to numbers that are not representable by a floating-point number type of any precision. In general, even without these operations, it is impossible to choose a fixed precision that is high enough to avoid all potential errors. In this section, we give an overview on various techniques that have been developed in order to create efficient number types that are able to guarantee exact decisions.

2.2.1 The Exact Computation Paradigm

In the early 1990s, Chee Yap and Thomas Dubé introduced the *Exact Computation Paradigm*, contrasting the previously predominant Fixed Precision Paradigm. [DY93; YD95]. An *exact number type* is characterized by two properties:

exact number type

1. Each number that can be constructed through its operations must have an exact representation in the number type.
2. All decisions made by the number type must be correct.

decision

By using the term *decision* we refer to any test on whether a number a represented by the number type is less than, greater than, or equal to another number b. Note that this is equivalent to determining the sign of $a - b$, hence it is sufficient to correctly determine the sign of a represented number. The main insight of Yap and Dubé is that making exact decisions is sufficient to solve the robustness problem in computational geometry. Each choice of an algorithm is based on a decision problem. If all decisions are correct and the input data is not already inconsistent, the algorithm will never make inconsistent choices. Bigfloat number types, as introduced in Section 2.1, cannot fulfill the first (and therefore the second) criterion as soon as division operations are involved. Quotients like $\frac{1}{10}$ (decimal) would need an infinite number of bits in the mantissa to be represented exactly by a binary floating-point number. Rational numbers can, however, be represented exactly as a combination of two integers.

2.2.2 Rational Exact Computation

Binary representations of an integer are always finite. Integer numbers can therefore be represented exactly by multiple-precision number types. If we allow the precision of a bigint to grow during an operation, integer operations such as addition, subtraction, multiplication and integer division can be performed exactly in accordance with the Exact Computation Paradigm. Rational numbers can be created by storing the numerator and denominator in separate bigints. In this form, there is no need for exact divisions on integer or floating-point values since they can be performed through a multiplication by the inverse. Multiplications, however, are naturally performed by multiplying numerators and denominators. Additions and subtractions are executed on the numerators after expanding the fractions to a common denominator.

In contrast to integer or floating-point number types, comparing the values of two exact rational number types requires a little more effort. In a naive way it can be done by temporarily expanding the stored fractions to a common denominator and comparing the numerators. This method requires two integer multiplications. The cost of an equality test can be reduced by maintaining the rationals in a normalized form, i.e. in a form where the greatest common divisor of the numerator and the denominator is one. The normalization can be done either after every operation or whenever a comparison is requested. It requires a reduction of the fraction by the greatest common divisor of numerator and denominator and therefore the computation of the greatest common divisor as well as two integer divisions. While expensive, normalization reduces the size of the number type and simplifies equality tests. If two fractions are normalized, they are equal if and only if both the numerators and the denominators are equal. It is, however, questionable, whether the benefits of normalization outweigh the additional cost associated with it [KLN91]. Furthermore, normalization is not sufficient to establish an order relation. Nevertheless, if two rationals are known to be unequal, the running time of their comparison can sometimes be improved by a stepwise computation and comparison of their continued fractions representations.

Since exact rational arithmetic is a fairly natural extension of exact integer arithmetic, many libraries mentioned in Section 2.1.3 provide implementations of exact rational arithmetic as well. Nevertheless, only few libraries implement non-trivial strategies to reduce its complexity. The rational numbers in `GMP` are always normalized. Other libraries, such as `LEDA`, leave the choice to normalize to the programmer. The `rational` type in `boost` normalizes the representation and additionally uses continued fractions to speed up comparisons [Moo19]. Despite these efforts, the increase in operand size as well as the high cost of comparisons make rational exact computation slow. Various studies report a running time increase by a factor of about 10^4 if primitive floating-point numbers are replaced by exact rationals in geometric algorithms without further optimization [KLN91; FW93a; Jai93]. In practice, rational arithmetic is therefore commonly used in combination with filter algorithms based on fixed-precision computations.

2.2.3 Floating-Point Filters

In general, through the usage of fixed-precision arithmetic the exactness of a result cannot be guaranteed. As described in Section 1.2, there are cases where inexact computation leads to wrong decisions and therefore to combinatorically incorrect results. In many cases, however, decisions taken on the basis of fixed-precision computations are still correct. Instead of directly computing the exact value of an expression, we can compute an upper and a lower bound to the value using a fixed precision number type. If zero is not part of the interval defined by these bounds, we can determine the sign of an expression without the need for exact computation. This technique is called *arithmetic filtering*. Upper and lower bounds for the value of an expression are obtained naturally through the use of *interval arithmetic*, where, instead of single approximations, a range of possible values is maintained and returned. This range is commonly represented by either two boundary points or a midpoint and a radius. The midpoint-radius representation has the advantage that only one value must be known at high precision (cf. Figure 2.4). An early usage of arithmetic filters is presented in the work of Michael Karasick, Derek Lieber and Lee Nackmann. In order to speed up their algorithm based on rational arithmetic, they use fixed-precision integer intervals to approximate the integers occurring during a sign computation [KLN91]. If an integer interval is not sufficient to make a decision, the interval would be recomputed at a higher precision. This process terminates at the latest when the used precision is sufficient to represent all values exactly.

interval arithmetic

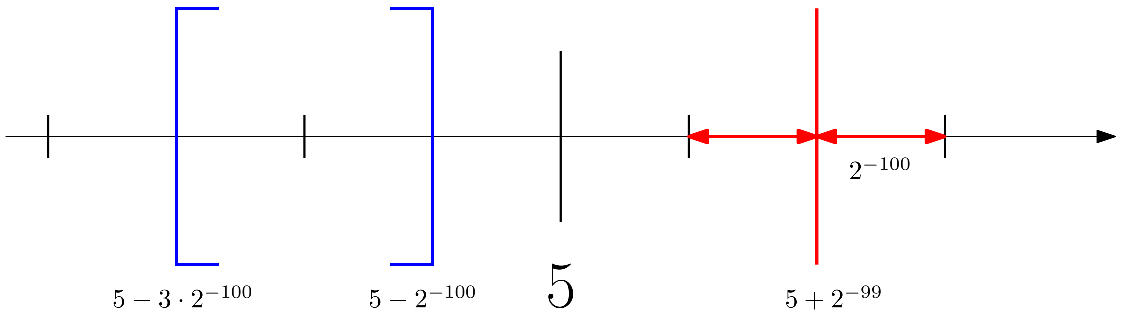


Figure 2.4: The endpoint representation (blue, left) and the midpoint-radius representation (red, right) demonstrated on two same-sized intervals near 5. For the endpoint representation, two multiple-precision numbers are needed to represent the boundaries. For the midpoint-radius representation, a single multiple-precision number is needed for the midpoint. The radius can be reasonably approximated by a power of two for small intervals.

An especially effective arithmetic filter can be constructed if interval boundaries are computed with fixed-precision floating-point number types. First proposed by Steven Fortune and Christopher van Wyk, *floating-point filters* are highly successful in reducing the need of exact computation [FW93a; DP98; BBP01]. Even with primitive floating-point types, most decisions occurring in geometric algorithms can be made exactly if the data is not seriously degenerated. Since operations on primitives are much faster than operations on multiple-precision number types (cf. Table 2.1, page 15), this property makes the application of floating-point filters very efficient.

floating-point filter

Similar to arithmetic filters based on integers, if the interval obtained through a floating-point filter does not contain zero, the sign of an expression can be determined without further computations. On the other hand, it cannot be decided whether the value of the expression is exactly zero in the opposite case. To guarantee exact computation, a fallback strategy, such as exact computation with rationals, is needed if the result of the filter is not decisive. As an intermediate step, if the precision of primitive types is not sufficient, bigfloats can be employed to successively increase the precision until a decision can be made or a certain threshold is reached. This concept is called a *cascading filter*. While primitive data types are almost guaranteed to produce negligible overhead,

cascading filter

filters based on multiple-precision arithmetic must be used with caution. A reasonable progression is achieved if the precision used for the interval arithmetic is doubled after each attempt. Since all operations have at least linear cost, this strategy ensures that the total cost of the filter mechanic does not exceed two times the cost of the computation at the highest precision, which in turn is at most twice as much precision as needed if a decision can be made through the filtering mechanism. It can be shown that this strategy is optimal if all implemented operations can be done in near-linear time [KR06]. For complex operations, however, a doubling of the expected computation time would be more appropriate [Hoe06c]. There are many libraries providing interval arithmetic that can be utilized for the implementation of a floating-point filter. Among others, both the `boost` and the `LEDA` library implement interval arithmetic on floating-point primitives [MPB19; Uhr17]. Interval arithmetic on multiple-precision floating-point types is provided by the `MPFI` library based on the bigfloat implementation in `MPFR` [RR05].

Remark 2.6. A technique related to floating-point filters is the *zero rewriting* approach of Kiyoshi Shirayanagi and Hiroshi Sekigawa [SS09]. In its basic version, zero rewriting uses interval arithmetic to decide whether a value is non-zero but, in contrast to classical filters, assumes that otherwise the value of the (sub-)expression is zero. Instead of verifying the result of the computation, it aims to verify the output of the algorithm and restarts the algorithm if the result is wrong. *Correct zero rewriting* on the other hand verifies each rewriting step separately by an exact computation. If the rewriting was incorrect, it determines a precision that is sufficiently high to solve the conflict and restarts the computation from scratch. However, zero rewriting techniques have not yet shown to be competitive compared to techniques based on classical floating-point filters [SW17].

For both simple and cascading filters, the program must be able to rerun the computation at least once. Karasick et al. require the host code to manually rerun the computation. Fortune & van Wyk do the same, but provide a pre-compiler that automatically inserts the necessary recomputation loops. A more comprehensive and ultimately more powerful strategy is based on the utilization of arithmetic expression dags.

2.2.4 Arithmetic Expression DAGs

Exact number types must at all times provide an exact representation to the number they currently represent (cf. Section 2.2.1). Exact rationals, as defined in Section 2.2.2, do this by managing two integer types with steadily increasing precisions. Therefore, the running time of each additional operation usually increases with the number of previous operations. However, in principle it is not necessary to compute the value of an operation in the moment where the operation is called. Instead, computing values is only necessary when a decision must be made. Delaying the actual computation up to this point yields several advantages since then the whole expression is known and algorithms can make an informed choice on what exactly has to be computed. This is called the *lazy* approach on evaluation. It is realized by storing the computation history in a graph structure [Ben+93a]. lazy expression dag **Definition 2.7.** An *(arithmetic) expression dag* is an ordered directed acyclic graph where each node either represents a number and has no children or represents an operation and has n (not necessarily disjoint) children, where n is the operation's arity. The nodes containing numbers are called the *leaves* or the *operands* of an expression dag. leaf, operand

Remark 2.8. While DAG is technically an acronym, we write terms like "expression dag" in lower case throughout this work in order to increase readability.

With an expression dag, computations can be saved in the dag representation as a new operator node pointing to the respective operands. Operator nodes can contain any operator that is defined on the number type. A leaf in an expression dag represents any number that is exactly representable in a chosen format. During an evaluation, parts of an expression dag may be converted to a single leaf if a more suitable exact representation is found. Figure 2.5 shows a very simple expression dag representing a rational number. Note that an exact rational number type can be interpreted as an expression dag with only one division node as operator where after each operation the expression dag is compressed.

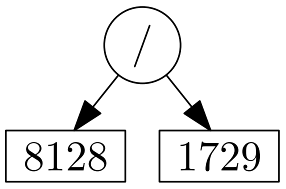


Figure 2.5: A rational number formed by two integers, depicted as an expression dag.

Remark 2.9. If not specified otherwise, we always assume that an expression dag is rooted, i.e., that it contains a distinguished root node from which all other nodes in the expression dag are reachable. Although in general an expression dag built during an algorithm is not necessarily rooted, a decision is always requested for a rooted subgraph. Hence, for most of this work we adhere to this local view of an expression dag. Furthermore, we do not strongly distinguish between the concept of an arithmetic expression and the concept of a rooted arithmetic expression dag. This is justified by the fact that there is a natural bijection between these concepts. Nevertheless, we still aim to use the most adequate term in a specific situation whenever readability is not negatively affected by the distinction.

Expression dags allow for the whole computation history to be stored in the number type. The actual evaluation of the operations can then be delayed up to the point where an exact decision is requested. A lazy evaluation opens the door for several optimization algorithms that require knowledge about the final expression. Most prominently, arithmetic filtering strategies can easily be realized on the generated graph structure. The first number type that implemented a floating-point filter based on lazy evaluation is the number type `LEA`. It builds an expression dag and uses a single floating-point filter on primitives before falling back to rational exact computation [Ben+93b]. Since then, implementations that require exact rational computation frequently delay the exact computation through the construction of an arithmetic expression dag in favor of filtering techniques [FNS04; ZXY16]. The class `lazy_exact_nt` of the geometrical algorithms library `CGAL` acts as a wrapper for arbitrary exact number types. It builds an expression dag and uses interval arithmetic on primitives to make decisions before a computation with the underlying number type is invoked [PF11].

2.2.5 Separation Bounds

When pairing a lazy evaluation with cascading floating-point filters one needs to decide at which point an exact computation should be used because increasing the precision of the number type is unlikely to lead to better results. This process requires a fair amount of (more or less) educated guessing. A more reliable approach can be pursued by computing a lower bound to the smallest value unequal to zero that is attained by any expression with similar structural properties.

Definition 2.10. Let E be an expression and let $\mathrm{value}(E)$ be the true value of E. Then a *separation bound* $\mathrm{sep}(E) \neq 0$ is a number, such that

separation bound

$$|\,\mathrm{value}(E)| < \mathrm{sep}(E) \Leftrightarrow \mathrm{value}(E) = 0$$

Finding a separation bound for an expression not only answers the question at which point increasing the precision is no longer useful, but it eliminates the need for a separate exact computation entirely. As soon as the computed error interval falls into the interval described by a separation bound, the value of the expression can only be zero (cf. Figure 2.6). Number types relying on separation bounds to make exact decisions are called *exact-decisions number types.*

exact-decisions nt.

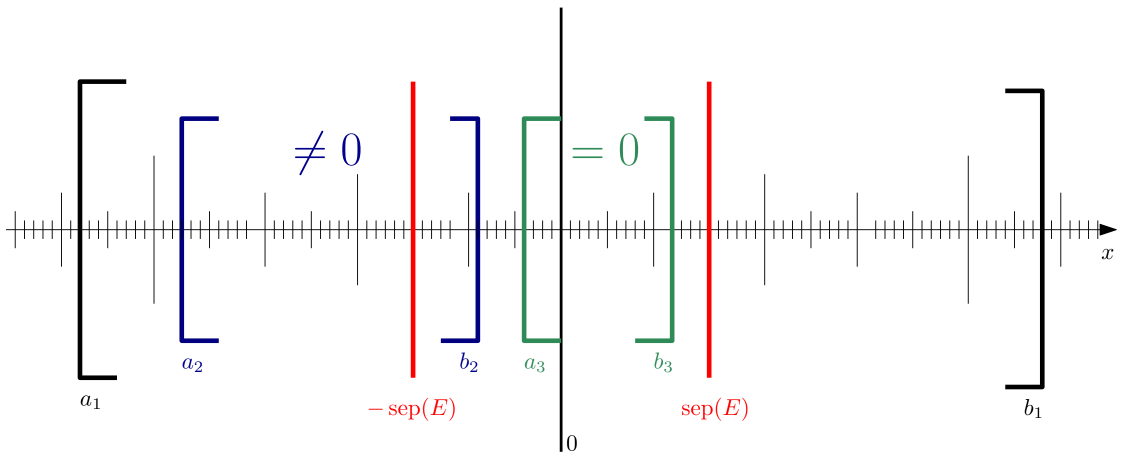


Figure 2.6: The three stages of a computation with cascading filters. The red lines represent a separation bound. The interval $[a_1, b_1]$ contains zero and values whose size is larger than the separation bound. In this case, it cannot be decided whether the value is zero and further iterations are needed. If the result of an iteration yields $[a_2, b_2]$, the interval does not contain zero and therefore $\mathrm{value}(E) < 0$. If the iteration yields $[a_3, b_3]$, each possible value for the expression is smaller than the separation bound and therefore $\mathrm{value}(E) = 0$.

Obviously, the optimal value for a separation bound is the value of the expression itself, or infinity if the value of the expression is zero. Computing this value is equivalent to an exact computation. Nevertheless, for many expressions a reasonably large separation bound can be computed much faster than an exact value. Furthermore, separation bounds can be found for values that are not exactly representable in floating-point or constructive

rational arithmetic at all. Most notably, all real algebraic numbers admit an efficient separation bound computation algorithm, although the resulting separation bound is not necessarily large enough to be of practical use. Separation bounds that admit an effective computation are called *constructive* separation bounds. Several efficient algorithms for the computation of separation bounds have been found [Sch09]. The best currently known bounds are based either on the computation of a bound for the Mahler measure of an algebraic number (cf. [Mah62]) or on the representation of an algebraic number as a quotient of two algebraic integers. We introduce two notations, which can be used to describe the set of operations that is covered by a constructive separation bound (cf. [Bur+09; LPY05]).

Definition 2.11. For $n \geq 0$ let $\alpha_0, ..., \alpha_d$ be algebraic numbers. Then the *diamond operation* $\diamond(j, \alpha_d, ..., \alpha_0)$ returns the j-th real root of the polynomial

$$P(X) = \sum_{i=0}^{d} \alpha_i X^i$$

Definition 2.12. Let $\overset{u}{-}$ refer to a negation, let $\mathbb{A}$ refer to the set of all algebraic numbers and let

$$\Omega = \{\overset{u}{-}, +, -, \cdot, /\} \cup \{\sqrt[d]{\bullet} \mid d \in \mathbb{N}\}$$
$$\Omega^* = \Omega \cup \{\diamond(j, \bullet_d, ..., \bullet_0) \mid j, d \in \mathbb{N}\}$$

radical

real algebraic

For $S \subseteq \mathbb{A}$ we call the set of expressions consisting exclusively of numbers $x \in S$ and operators from Ω the *radical expressions* over S. Furthermore, we say that a number is a *radical number* if it is the value of a radical expression over $\mathbb{Z}$. Likewise, we call the expressions over S with operators from Ω^* the *real algebraic expressions* over S and the numbers created by algebraic expressions over $\mathbb{Z}$ the *real algebraic numbers*.

In literature, the radical expressions over $\mathbb{Z}$ are sometimes referred to as “constructible” expressions. However, the radical numbers form a proper superset of all (compass and straightedge) constructible numbers, which, for example, are not closed under the application of third roots. In order to avoid misunderstandings, we use the term *radical* instead. Note furthermore that the real algebraic numbers indeed form the set of all algebraic numbers in $\mathbb{R}$.

Measure-Based Separation Bounds

In his work on the identification of algebraic numbers, Maurice Mignotte describes a separation bound based on the “size” of an algebraic number and suggests that the Mahler measure of an algebraic number could be a viable choice for a size function [Mig82]. Several years later, Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn and Stefan Schirra show that the described bound admits an efficient computation. They call the resulting constructive separation bound the *degree-measure bound* [Bur+00]. Let $\alpha \neq 0$

be an algebraic number, let $\deg(\alpha)$ be the algebraic degree of α and let $M(\alpha)$ be the Mahler measure of α. Then the size of α is bounded as

$$|\alpha| \geq \frac{1}{M(\alpha)}$$

Now assume α is the value of the expression represented by an expression dag E, then the degree-measure bound algorithm inductively computes an upper bound on $M(\alpha)$ together with an upper bound on $\deg(\alpha)$ by computing appropriate bounds for the leaves of E and maintaining them for the subexpressions while applying operations. The algorithm provides induction steps for all operations $\circ \in \Omega$, hence a separation bound can be found for all radical expressions. The bound for $M(\alpha)$ was later improved by Chen Li and Chee Yap [LY01]. Another, conceptually different, improvement on the measure was proposed by Hiroshi Sekigawa [Sek04]. We call the bound resulting from Sekigawa's improvement the *Sekigawa bound*. In their paper, Li and Yap additionally introduce a new separation bound algorithm involving the computation of both degree and measure bounds, but based on another inequality. For an algebraic number $\alpha \neq 0$ let $\mu(\alpha)$ be the maximum size of a conjugate of α and $\text{lead}(\alpha)$ be the size of the leading coefficient of the minimal polynomial of α. Then

$$|\alpha| \geq \frac{1}{\mu(\alpha)^{\deg(\alpha)-1}\,\text{lead}(\alpha)} \tag{2.1}$$

The separation bound resulting from this inequality is known as the *conjugate bound* or the *LY bound*. Both the improved degree-measure bound and the LY bound support radical expressions over real algebraic numbers, given by a polynomial, as input. In 2006, Sylvain Pion and Chee Yap introduced a so-called generic k-ary method, which improves basically any constructive separation bound by taking advantage on a k-ary input format. In particular, they apply their technique on the (improved) degree-measure bound algorithm, calling the resulting separation bound the *k-ary measure bound* [PY06].

Quotient-Based Separation Bounds

Alongside a precise description of the degree-measure bound, Burnikel et al. introduce their own separation bound algorithm [Bur+00]. Their algorithm can loosely be described as bounding the size and the algebraic degrees of (the conjugates of) the algebraic integers α_1, α_2 occurring in an equivalent "algebraic rational" expression, i.e., an expression of the form $\frac{\alpha_1}{\alpha_2}$. Considering $\alpha_2 \leq \mu(\alpha_2)$ and $\text{lead}(\alpha_1) = 1$, a separation bound for the algebraic number $\alpha = \frac{\alpha_1}{\alpha_2}$ can then be obtained similarly to (2.1) by

$$|\alpha| \geq \frac{1}{\mu(\alpha_2)\mu(\alpha_1)^{\deg(\alpha_1)-1}}$$

The associated algorithm provided in the paper can be applied to any radical expression. Its resulting separation bound is called the *BFMS bound*. The bound was later improved in a paper of Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra and

Susanne Schmitt, who observe that the algebraic degree bound of α_1 is quadratic in the algebraic degree bound of α and adjust the iteration such that the degree bounds match [Bur+09]. Furthermore, they extend the original algorithm by the diamond operator, thereby supporting all real algebraic expressions. The resulting separation bound is called the *BFMSS bound*.

Beside their improvement of the degree-measure bound, Pion and Yap apply the k-ary method on the BFMSS bound and call the result the BFMSS[k] bound [PY06]. The induction step for the diamond operator in the k-ary bound was later improved by Susanne Schmitt [Sch04]. The rules of the BFMSS[2] bound for radical expressions are depicted in Table 2.7. The improved degree-measure bound, the LY bound and the BFMSS bound are principally incomparable, i.e., for either of them there are expressions where they lead to the best separation bound. Apart from that, the BFMSS bound dominates most other separation bounds, both theoretically and practically [Mör15a].

Expression E	v	u	l
$n2^m$ $(n, m \in \mathbb{Z})$	m	$\lvert n\rvert$	1
$-E_1$	v_1	u_1	l_1
$E_1 \pm E_2$	$\min(v_1, v_2)$	$2^{v_1-v}u_1l_2 + 2^{v_2-v}u_2l_1$	l_1l_2
$E_1 \cdot E_2$	$v_1 + v_2$	u_1u_2	l_1l_2
E_1/E_2	$v_1 + v_2$	u_1l_2	l_1u_2
$\sqrt[d]{E_1}$ $(2^{v_1}u_1 \geq l_1)$	$\lfloor v_1/d \rfloor_0$	$\sqrt[d]{2^{v_1-dv}u_1l_1^{d-1}}$	l_1
$\sqrt[d]{E_1}$ $(2^{v_1}u_1 < l_1)$	$\lfloor v_1/d \rfloor_0$	u_1	$\sqrt[d]{2^{v_1-dv}u_1^{d-1}l_1}$

Table 2.7: The iteration steps for the parameters v, u, l of the BFMSS[2] bound. The operation $\lfloor\ \rfloor_0$ used in the root cases indicates rounding toward zero. The parameters u and l denote upper bounds for $\mu(\alpha_1)$ and $\mu(\alpha_2)$, where $\alpha = 2^v\frac{\alpha_1}{\alpha_2}$ is the value of E. For $v = 0$, the iteration steps for u and l yield the BFMSS bound. A separation bound is given as $|\alpha| \geq \frac{2^v}{u^{\deg(\alpha)-1}l}$. A bound to the algebraic degree $\deg(\alpha)$ is usually computed separately as the product of the degrees of all root operations occurring in E.

2.2.6 Accuracy-Driven Evaluation

When evaluating an expression, number types like `LEA` choose a certain precision for the underlying floating-point type and compute an approximation together with an error bound. If the error bound is not sufficient to determine the sign of the expression, they switch to a (rational) exact computation. Through the use of separation bounds, an upper bound to the accuracy needed to determine the sign of an expression is known. Hence, with a cascading floating-point filter the precision can be gradually increased until a decision can be made (cf. Section 2.2.3). We call this strategy an *adaptive precision evaluation*. However, this approach does not take into account that at different nodes a precision increase contributes differently to the overall error bound. If, for example, two numbers with very different sizes are added, an increase in relative accuracy for the smaller number does not have much effect on the accuracy of the result, if any effect at

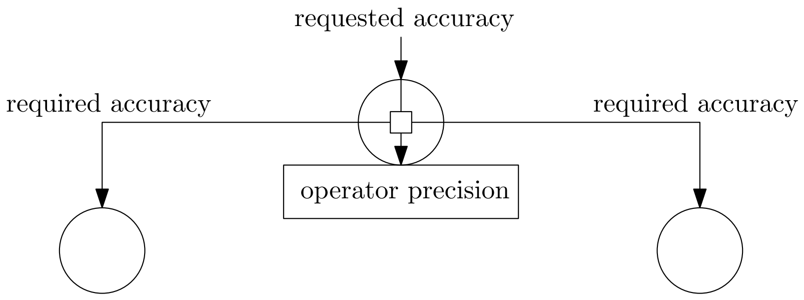


Figure 2.8: A basic depiction of the accuracy propagation step during an accuracy-driven evaluation for a binary operator node. An incoming requested accuracy is split into three required accuracy bounds. The accuracy bound for the operation at the given node is translated into an operator precision (cf. Section 2.1.1). The child nodes are evaluated recursively before performing the operation.

all. Therefore, the precision of operations forming the bigger value should be much higher than the precision of operations forming the smaller value. Following a lazy evaluation approach, we can compute the precision that is required at each node to guarantee a certain overall error bound before an actual evaluation takes place. To determine those precisions, at each node we choose an operator precision and accuracies for the child nodes such that the target accuracy for this node is guaranteed if both the errors at the child nodes can be bounded accordingly and the operation is performed at the chosen precision (cf. Figure 2.8). By recursively evaluating the child nodes before computing the actual operation, the node is evaluated to the requested accuracy. This strategy is called *accuracy-driven evaluation*. accuracy-driven

Remark 2.13. The concept of accuracy-driven evaluation was first described by Thomas Dubé and Chee Yap. In their work they coined the term “precision-driven computation” for this strategy [DY93]. In the opinion of the author, the usage of this term is misleading since the main advantage of the new strategy is that the precision of a number type is determined by variable accuracy requests. The process is therefore driven by the choice of accuracies. In fact, one might argue that the term “precision-driven” is more suitable for the adaptive precision strategy described above since in this case the computation is driven by the choice of precisions, i.e., the resulting accuracy is predetermined by the initially chosen precision. Consequently, in this work, we exclusively use the term “accuracy-driven” to refer to the strategy of Dubé and Yap.

To compute the required accuracies at the child nodes, a lower and an upper bound for their size must be known. For this, performing a fixed-precision computation with low precision is sufficient, except for two situations. First, the denominator of a division must always be non-zero. Second, the operand of a root operation must always be non-negative. In these cases the sign of the respective operands must be computed recursively before the main evaluation starts. Aside from the initial fixed-precision evaluation, a cascading floating-point filter can and should be implemented together with accuracy-driven evaluation. Separation bounds can get very small, especially when root

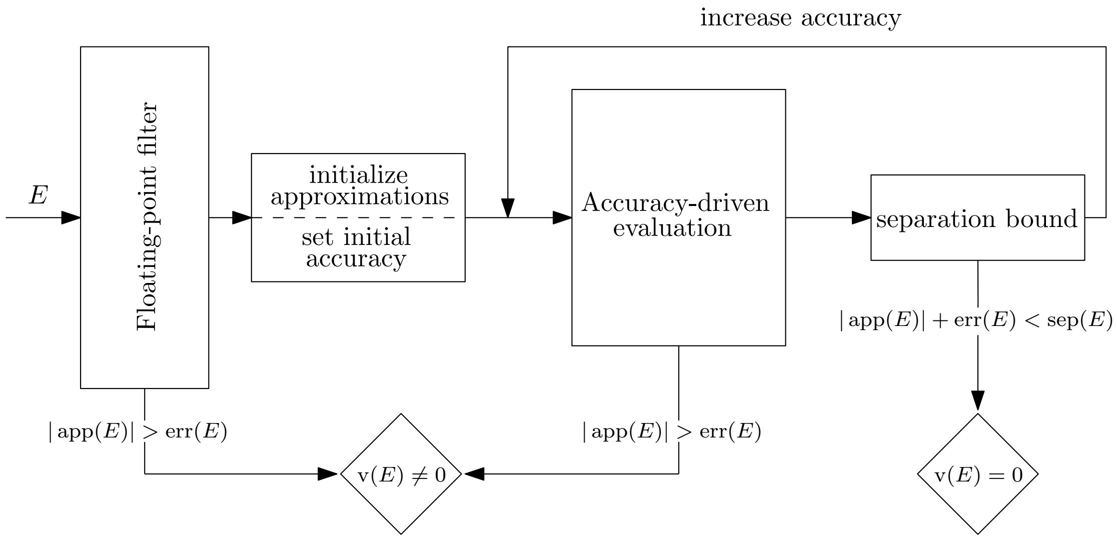

increase accuracy
E
Floating-point filter
initialize approximations
set initial accuracy
Accuracy-driven evaluation
separation bound
$|\operatorname{app}(E)| + \operatorname{err}(E) < \operatorname{sep}(E)$
$|\operatorname{app}(E)| > \operatorname{err}(E)$
$|\operatorname{app}(E)| > \operatorname{err}(E)$
$\operatorname{v}(E) \neq 0$
$\operatorname{v}(E) = 0$

Figure 2.9: The principal stages of a decision process as implemented in exact-decisions number types. The current approximation and the current error bound for an expression E are denoted as $\operatorname{app}(E)$ and $\operatorname{err}(E)$. The actual value of E is denoted as $\operatorname{v}(E)$. Before any data is initialized, a floating-point filter on primitives is used. If the initial filter is insufficient, a cascading filter mechanism based on accuracy-driven evaluation is executed until a decision can be made, either directly or by hitting a separation bound.

operations are present. If the value of an expression is non-zero, in most cases its size is much larger than the separation bound associated with the expression. Similarly to the strategy described in Section 2.2.3, we can start by requesting a low accuracy and increase the requested accuracy until we can separate the value from zero or reach the separation bound. In contrast to an adaptive precision evaluation, we do not increase the precision of the computation but the accuracy of the result. The basic design of an accuracy-driven number type is shown in Figure 2.9. Note that, although not depicted, the initialization may require several calls to the complete decision process for subexpressions.

The number type `real` from the `LEDA` library and the number type `Expr` from `CORE` were the first to use accuracy-driven evaluation to make exact decisions. While different at the time of their creation, newer versions of these number types are largely identical and differ only with respect to minor design choices. We briefly describe both libraries.

The `LEDA` Library

The Library of Efficient Data Types and Algorithms (`LEDA`) is a collection of algorithms and data types for combinatorial and geometrical computing. It is in development since the late 1980s with the goal of making complex efficient algorithms available for widespread practical application [MN89; MN99]. For exact computation, `LEDA` defines number types for integers, rationals and floating-point numbers. The most powerful number type in `LEDA` for exact computation, the number type `real`, facilitates accuracy-driven evaluation to guarantee exact decisions for arbitrary radical numbers [Bur+95; BMS96]. The original version of the data type used the degree-measure bound together with a cascading floating-point filter. Newer versions employ the BFMSS[2] bound and extend the implementation to include the diamond operator and therefore to support all real algebraic numbers [Bur+04; Sch05].

The `CORE` Library

The main purpose of `CORE` is to provide a library that can be included in standard C++ programs to make all decisions in the program more precise or even exact. Initially proposed in 1998, a first version of the library was introduced one year later [Yap98; Kar+99]. In contrast to `LEDA`, the `CORE` library does not provide additional algorithms or data structures but consists exclusively of various number types. It defines four levels of accuracy. At the first level, only primitive number types are used. The second level replaces those primitives by multiple-precision number types (cf. Section 2.1). The third level guarantees exact computation. Finally, the fourth level allows for a mixture of those accuracy levels. For the third level, a number type called `Expr`, based on the number type `RealExpr` by Kouchi Ouchi, is used [DOY94; Ouc97]. In its first version, `Expr` supported rational arithmetic as well as square root operations, based on the so-called degree-length bound, an early constructive separation bound that can be shown to be inferior to the degree-measure bound [LY01]. Later versions of `CORE` add a floating-point filter, use the BFMSS[2] bound and extend the range of supported expressions to all real algebraic expressions [Yu+10]. Furthermore, they employ a partly modularized implementation scheme, which can be utilized to add further operations or exchange the separation bound algorithm.

2.3 Effective Number Types

Largely independently of exact number types, the theory of effective number types has evolved. It dates back to the notion of *effective calculability* of functions, which describes those functions that are "intuitively" computable. In the 1930s, three different computational models were proposed in order to formalize the concept of effective calculability. Kurt Gödel developed the notion of recursive functions based on a suggestion of Jacques Herbrand and refined by Stephen Kleene, whereas Alonzo Church introduced the λ-calculus and Alan Turing his Turing machines. All of these models were proven to be equivalent [Chu36; Tur37; Ros39]. It is widely assumed that the set of functions over natural numbers that are effectively calculable is equal to the set of functions that can be described by Turing machines or, equivalently, any of the other models. This assumption is known as the Church-Turing thesis. The definition of computability through Turing machines is equivalent to computability in the RAM machine model. In the real RAM model, the definition of computability is extended to real numbers by allowing the application of analytic functions as well as comparisons on arbitrary real numbers (cf. Section 1.1). While this model allows a very intuitive development of theoretical algorithms, it is not immediately clear if and how the postulated operations can be implemented on a real machine, which produces a gap that is partly filled by the development of exact number types. Alan Turing calls a real number *computable* if it admits an effective approximation algorithm, i.e, an algorithm on natural numbers, realized by a Turing machine, that computes an arbitrarily good (rational) approximation for said number [Tur37]. Many authors proposed machine models that extend the concept of computability based on Turing's definition in order to enable the use of analytical methods while preserving the general notion of effectiveness [Grz55; Maz63; Kus84; PR89; Ko91; Gia99]. A comparison of these, mostly similar, models, paired with a rather intuitive machine model is given by Klaus Weihrauch [Wei00]. The field born from these efforts is known as *computable analysis*.

computable

Weihrauch allows the representation of a real number r by a sequence $(I_n)_{n\in\mathbb{N}}$ of shrinking intervals with rational endpoints and $r = \bigcup_{n=1}^{\infty} I_n$. He calls a function *computable* if there is a multi-tape Turing machine that from time to time produces a new interval of an interval sequence representation for the result while reading the interval representations of its parameters. This theory is called the "Type-2 Theory of Effectivity" (TTE) in order to distinguish it from "Type-1" computability. Following the definition of Weihrauch, we say that a real number r is called *effective* if it is TTE-computable. An *effective number type*, consequentially, is a number type that provides representations for the effective (real) numbers and supports basic TTE-computable functions. Effective numbers contain the (real) algebraic numbers and important transcendental numbers, such as π and e. Furthermore, they are closed under the application of all elementary functions, such as sin, cos, exp and the natural logarithm.

effective

TTE-computable functions are fundamentally incomparable to the functions that can be computed following the real RAM model. Due to the inclusion of all algebraic numbers, effective numbers are, however, more general than the numbers representable by the exact number types introduced in the previous section. It was shown that a large subset of the effective numbers can be compared to zero if Schanuel's conjecture is true [Ric97]. Nevertheless, in general it is not possible to decide whether an effective number is zero [Hoe06c]. So, while authors commonly refer to their implementations of effective number types as using "exact arithmetic", they are not exact in the sense of the Exact Computation Paradigm (cf. Section 2.2.1), although some of them demand at least the weaker property that the number type is able to correctly identify the order of every two distinct effective numbers. Nevertheless, similar techniques were developed for effective number types as they were developed for exact number types. Effective number types are usually based on expression dags, which are evaluated either top-down/a priori (accuracy-driven) or bottom-up/a posteriori (with adaptive precision) [Bla02]. Note that the accuracy-driven approach comes naturally to effective number types since it reflects the definition of effectiveness. In an attempt to enable zero-testing, witness conjectures are proposed, which, if true, would result in separation bounds [Hoe97; Ric00; Hoe01; RE03]. Although some of the conjectures have been disproven, others remain open problems [Hoe06b; RE06]. Joris van der Hoeven and John Shackell were able to prove a doubly-exponential separation bound for power series [HS06].

A large amount of implementations of effective number types are described in literature. Virtually all of them are based on the definition of Weihrauch or an equivalent one. Especially in the early days of computable analysis, several functional implementations were proposed [BC90; BES02]. In contrast to later implementations, they did not explicitly use expression dags as they could be replaced by backtracking in the functional structure. Similarly, a package designed by Norbert Müller employs long jumps in order to do recomputations at higher precisions [Mül00]. Müller's package is unique in the sense that it claims to simulate a real RAM. It uses bottom-up evaluation and filter mechanisms to gradually increase the precision up to a certain limit. Although described as using exact arithmetic, neither equality nor inequality tests can be performed exactly. Aside from the functional approach, several other representations of effective numbers exist [GL00]. David Berthelot and Marc Daumas represent numbers by a sequence of floating-point intervals [BD97]. Starting with their work, most subsequent effective number types employ interval arithmetic paired with expression dags, albeit Berthelot and Daumas themselves still used backtracking to increase the precision.

Keith Briggs was the first to implement and to demand that exact arithmetic must be able to handle inequality tests correctly if they are positive [Bri06]. At least up to this point it can be argued that all previous effective number types are closer to multiple-precision number types than to exact number types. Branimir Lambov implements a number type supporting the computation of limits and comparisons which are guaranteed to be correct but may not return [Lam07]. Lambov implements both a bottom-up and a top-down strategy and switches between them to increase the performance. Various implementation choices are discussed in Joris van der Hoeven's number type, albeit leaving open the final decision on how to implement comparisons [Hoe06c]. In 2010, Yong Li and Jun-Hai Yong presented a very limited number type with an attempt to a balanced top-down evaluation but without mentioning comparisons [LY07].

3 A Configurable Expression DAG Policy for Real_algebraic

Various experiments are performed throughout this work in order to assess the practical behavior of the proposed algorithms. All of these experiments use the exact-decisions number type `Real_algebraic`, which is perfectly suited for scientific testing due to its generic nature. In this chapter, we introduce an extension to `Real_algebraic` that modularizes the node data type of the underlying expression dag, one of its central components. In Section 3.1, we briefly describe the general concept and important mechanics of `Real_algebraic`. In Section 3.2, we describe the new modular node type and provide an overview on its modules.

3.1 The Number Type Real_algebraic

The number type `Real_algebraic` was introduced in 2010 by Marc Mörig, Ivo Rössling and Stefan Schirra [MRS10; RA15]. `Real_algebraic` is an exact-decisions number type based on `leda::real`. It implements accuracy-driven evaluation on an expression dag using the same basic strategies as `leda::real`, but it makes each component interchangeable by the use of generic programming. Parts that are largely independent, such as floating-point filters or the bigfloat number type used during the evaluation, can be exchanged easily by the user. For each component, an abstract *concept* is defined, which describes the purpose and the interface for a component. A class that satisfies the conditions imposed by a concept is called a *model* of the concept. These models may again define new concepts that must be provided by the user. A number type is then defined by a collection of models. We call such a collection a *configuration* of `Real_algebraic`.

concept

model

configuration

There are five main concepts constituting `Real_algebraic`. To emphasize the fact that they describe a behavioral aspect of a larger type, they are called *policies*. We give a brief description for each of those policies.

policy

3.1.1 LocalPolicy

Building an expression dag is expensive compared to a simple floating-point calculation with primitives. Sometimes operations on primitives lead to the exact result. Sterbenz' lemma, for instance, states that additions and subtractions on floating-point numbers a and b are always exact if $0.5 \leq \frac{a}{b} \leq 2$ can be guaranteed [Ste73]. Furthermore, for additions, subtractions and multiplications, it is easy to check whether the result is exact since the error of a floating-point operation can be represented exactly with another same-sized floating-point number type [Dek71]. With the same techniques, expressions consisting exclusively of these operations can be represented exactly as a sum of floating-point numbers with or without overlapping significant bits [She97; MS07]. If the significant bits do not overlap, i.e., the least significant bit of a larger floating-point number is always more significant than the most significant bit of a smaller floating-point number, the sign of the sum can be read off directly from the representation. Otherwise, it can be computed in reasonable time [RR99; MS07]. If implemented in an exact-decisions number type, these techniques can be utilized to delay or completely avoid the construction of an expression dag [Mör10].

The `LocalPolicy` permits the usage of alternative sign-determining methods while falling back to the creation of an expression dag if the sign computation fails. Albeit useful in an appropriate context, using an unsuitable local policy may have a negative impact on the performance of the number type. To avoid inconsistencies, we do not use a local policy in this work.

3.1.2 FilterPolicy

In `Real_algebraic`, an exchangeable floating-point-filter, designed to work with interval arithmetic on primitive types, is used (cf. Section 2.2.3). The details on the underlying interval arithmetic are specified by its `FilterPolicy`. During the construction of the expression dag, an interval for the value of the expression is determined. When a sign should be computed, the floating-point-filter is checked before initializing the bigfloat data. The interval computed by the floating-point filter interacts with the subsequent accuracy-driven evaluation in both directions. Initial approximations, needed for the computation of error bounds and separation bounds, are obtained from the filter if a finite interval representation exists. Conversely, approximations computed during the evaluation are used to adjust the floating-point interval. `Real_algebraic` currently supports the usage of either `boost` or `LEDA` interval arithmetic as floating-point filter.

3.1.3 ApproximationPolicy

During an accuracy-driven evaluation, approximations are computed (and stored) for each subexpression, i.e., for the expression at each node of the underlying expression dag. The `ApproximationPolicy` defines which bigfloat number type is used in `Real_algebraic`. Furthermore, it defines the standard type used for representing the exponent of a bigfloat. Contrary to its name, the `ApproximationPolicy` not only influences the computation of approximations, but also the computation of error bounds, which are, depending on the context, represented by bigfloats or by their exponents. Currently, models are provided for two different bigfloat number types, for the type `mpfr_t` based on the GNU multiple-precision library and for the `bigfloat` type from the `LEDA` library (cf. Section 2.1.3).

3.1.4 SeparationBound

`Real_algebraic` can be used with several different separation bound strategies by changing the respective policy. A model of the `SeparationBound` policy must provide a method returning a zero separation bound for the subexpression at any node in the expression dag when given an upper bound to the algebraic degree of the subexpression's value. For this, the method may rely on parameters that are computed inductively during the initialization. In particular, the model must provide methods for each implemented operation that are called whenever a node is initialized and that are given the separation bound data stored at the operands. Models are given for the degree-measure bound, the Sekigawa bound, the LY bound and the BFMSS bound with and without the extension for k-ary inputs (cf. Section 2.2.5). As a side product of this work, a generic model for a simultaneous use of any combination of the other models is available, which returns the largest separation bound that can be found by any of the chosen strategies.

3.1.5 ExpressionDagPolicy

The `ExpressionDagPolicy` provides the expression dag itself as well as the accuracy-driven evaluation. A valid `ExpressionDagPolicy` must provide methods to process all basic arithmetic operations, to compute the value of an expression to a given accuracy and to determine the sign of an expression. It is supposed to do so by creating an expression dag and performing accuracy-driven evaluation on it, using the concepts defined in the `FilterPolicy`, the `ApproximationPolicy` and the `SeparationBound`. While the underlying concept of the `ExpressionDagPolicy` enables a global evaluation process, all provided models operate locally. In these models, the main functionality is encompassed in a `node` class, representing a single node of the expression dag. Implementations of a basic dag node, called `sdag_node`, and a dag node copying the evaluation strategy used in `LEDA` are available. However, the two node types differ only with respect to minor implementation details.

3.2 The Class configurable_dag_node

The `ExpressionDagPolicy` in `Real_algebraic` allows for an exchange of the evaluation strategy as a whole. However, different evaluation strategies often require only a small change in the implementation. Furthermore, most changes are conceptually independent, leading to an exponential number of possible combinations. Consequently, we aim to create a new generic model of the `ExpressionDagPolicy`, which allows for an independent application of these changes. Central to the standard implementation of an `ExpressionDagPolicy` is the underlying node class, which both represents the root node of an expression dag and contains the evaluation mechanism. We propose a new node class called `configurable_dag_node` based on a set of new policies, enabling a more modular configuration of the evaluation process. The new node class is designed with three goals in mind:

1. The configurable node class must resemble the class `sdag_node` as closely as possible if the base set of policies is chosen. In particular, making a part of it generic must not have a negative effect on the performance of the base configuration.

2. Policies should be based on an intuitive concept. It should be intuitively clear for both the user and the developer which part of the node is affected by which policy.

3. Policies should be as independent as possible. The choice for one policy should not depend on the choice made for another policy.

Those three goals are to some degree in conflict with each other. If a conflict arises, they are prioritized top-down. Being able to closely simulate the `sdag_node` is vital for experimental results to be meaningful. An intuitive concept and independence of modules are important to make the code maintainable as well as extendable in the future. Those goals, however, must often be sacrificed in order to keep the exact structure of the reference class intact. In future versions of `Real_algebraic` the first criterion might be loosened, resulting in a more polished version with respect to maintainability.

There are four main aspects for which exchangeability is established by the new node class, namely preprocessing, evaluation, separation bound computation and error bound handling. In the following sections, we give an overview on the relevant concepts associated with each of these aspects.

3.2.1 Preprocessing

During the instantiation of a node class, usually only the structural information and the floating-point filter is initialized. The structural information of the expression dag can be exploited to optimize the evaluation process by implementing preprocessing strategies, which are executed right before the expression dag is evaluated. If cost-heavy member data, such as the bigfloats needed for the approximation, are not initialized before the node is evaluated for the first time, those strategies can make changes in the structure of the expression dag at low cost. The development of good preprocessing strategies is a key component in the handling of large expression dags. We establish a new concept called the `RestructuringPolicy`. Classes fulfilling this concept must provide a `restructure` method, which is called at every node, right before its first evaluation. The classes available for preprocessing are shown in orange in Figure 3.1. In Chapter 4, the algorithms behind the depicted restructuring strategies, except for the obvious `No_restructuring`, are described in detail.

3.2.2 Evaluation

At the core of `Real_algebraic` lies an accuracy-driven evaluation scheme as described in Section 2.2.6. A natural way of implementing accuracy-driven evaluation at a dag node is to compute error bounds for its children, recursively evaluate the children and, after that, execute the operation at the node using the approximations computed at the child nodes. We call this the *recursive evaluation strategy*.

recursive evaluation

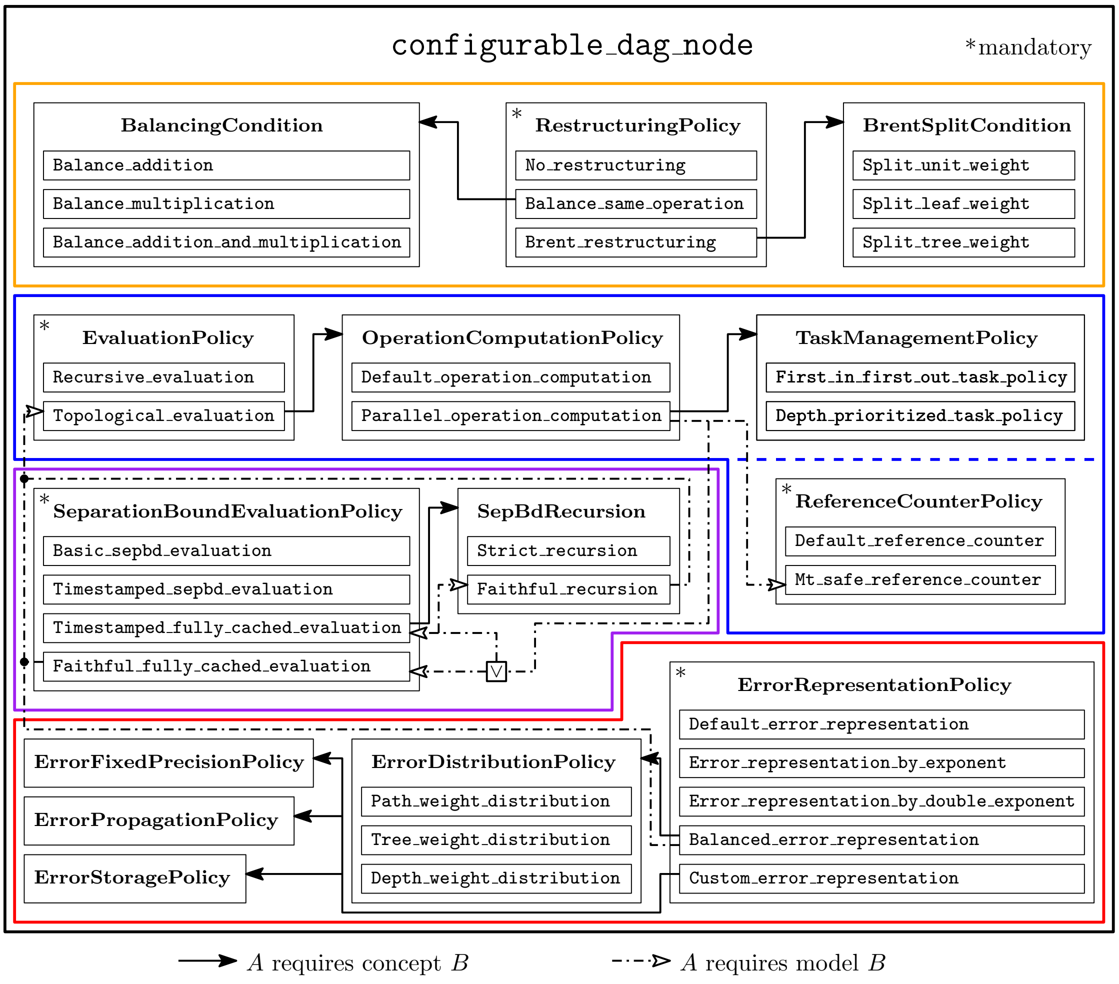

configurable_dag_node
*mandatory
BalancingCondition
Balance_addition
Balance_multiplication
Balance_addition_and_multiplication
* RestructuringPolicy
No_restructuring
Balance_same_operation
Brent_restructuring
BrentSplitCondition
Split_unit_weight
Split_leaf_weight
Split_tree_weight
* EvaluationPolicy
Recursive_evaluation
Topological_evaluation
OperationComputationPolicy
Default_operation_computation
Parallel_operation_computation
TaskManagementPolicy
First_in_first_out_task_policy
Depth_prioritized_task_policy
* ReferenceCounterPolicy
Default_reference_counter
Mt_safe_reference_counter
* SeparationBoundEvaluationPolicy
Basic_sepbd_evaluation
Timestamped_sepbd_evaluation
Timestamped_fully_cached_evaluation
Faithful_fully_cached_evaluation
SepBdRecursion
Strict_recursion
Faithful_recursion
∨
* ErrorRepresentationPolicy
Default_error_representation
Error_representation_by_exponent
Error_representation_by_double_exponent
Balanced_error_representation
Custom_error_representation
ErrorFixedPrecisionPolicy
ErrorPropagationPolicy
ErrorStoragePolicy
ErrorDistributionPolicy
Path_weight_distribution
Tree_weight_distribution
Depth_weight_distribution
A requires concept B
A requires model B

Figure 3.1: An overview on the concepts contained in the definition of `configurable_dag_node` as well as on the implemented models of these concepts. The concepts can roughly be categorized into preprocessing (orange), evaluation (blue), separation bound computation (purple) and error handling (red). Five of the concepts must be defined when using `configurable_dag_node`, the remaining concepts are only needed for specific models.

Although natural, the recursive evaluation strategy may lead to unexpected runtime behavior [MS15]. If the expression dag contains common subexpressions, the running time of the evaluation heavily depends on the evaluation order of the dag nodes. If the evaluation order is unfortunate, already evaluated subexpressions may need to be re-evaluated later with a higher requested accuracy. In the worst case, this leads to a quadratic number of re-evaluations.

Definition 3.1. Let G be a directed acyclic graph. A *topological sorting* of G is a total ordering of the nodes in G, such that for two nodes u, v where v lies in the subgraph of u it follows that $u < v$. topological sorting

To preclude inconsistencies caused by the evaluation order, a topological sorting of all nodes in the expression can be computed [MS15]. With a topologically sorted list of nodes, the highest accuracy needed at each node can be efficiently determined before any evaluation takes place. By this, it is guaranteed that each node is evaluated at most once. We call this approach the *topological evaluation strategy*.

topological evaluation

In the `configurable_dag_node`, the evaluation strategy is made interchangeable through the use of the `EvaluationPolicy` concept. The concept requires a class to implement a method called `guarantee_error_bound`, which computes the value of a given node v up to a certain requested accuracy. It is expected that the relevant error bounds are retrieved through methods provided by the `ErrorPropagationPolicy`, which is described in detail in Section 3.2.4. Classes fulfilling the specifications are provided for both recursive and topological evaluation. In the class `Topological_evaluation`, the above-mentioned method starts by computing a topologically ascending list of all nodes in the subexpression rooted at v and afterwards operates exclusively on the acquired list without any further recursions. In a first step, the necessary error bounds for all nodes are computed by traversing the list backwards. After that, the operations are computed according to a concept called `OperationComputationPolicy`. Computing the highest required accuracy at each node before computing the new approximations is not only useful to guarantee consistency, but it also enables a parallel computation of the associated bigfloat operations. The operations can be computed either in the preset order of the list or they can be processed such that topologically independent nodes are computed in parallel. The `OperationComputationPolicy` reflects this choice by providing a method that, given the topologically sorted list, calls the evaluation function on each node according to their topological order. In order to guarantee multithread-safety, modifications with respect to the representation of the reference counter and the separation bound computation strategy are required. Furthermore, different strategies on task prioritization may be employed for the parallel operation computation. The implementation of parallelization is described in more detail in Chapter 5. The associated concepts and classes are depicted in blue in Figure 3.1.

3.2.3 Separation Bound Computation

algebraic degree

To determine the sign of an expression, a separation bound is computed during the evaluation (cf. Section 3.1.4). The *algebraic degree* of a number x is the smallest degree of a polynomial that evaluates to zero at x. To get a separation bound for an expression that may contain root operations, an upper bound to the algebraic degree of the value of the expression must be found. The algebraic degree can be bounded by the product of the degrees of all root operations occurring in the expression. When a separation bound is needed for a subexpression, the default version of `Real_algebraic` traverses all nodes in the subexpression and multiplies their degree. This traversal can get expensive. If separation bounds need to be computed for all nodes during an evaluation, a linear traversal of each subexpression leads to a quadratic running time. Moreover, due to the access to the child nodes during the traversal, evaluating nodes is not multithread-safe. Although the access to the degrees of the nodes is read-only, guarding flags must be set at each node during the traversal to prevent common subexpressions from being visited more than once. Those flags may then be corrupted during a parallel evaluation.

Despite the obvious downside of continued traversals, it is not obvious how to avoid them. If the degree bounds are computed in a simple bottom-up manner from the bounds of its children, the resulting degree bound can be higher than with a traversal if nodes with high degree appear in more than one subexpression. A slightly higher degree bound, however, can already induce a considerably worse separation bound. In the class `configurable_dag_node`, the methods that are used to compute the algebraic degree bound and to trigger update procedures of the `SeparationBound` class are summarized in a concept called `SeparationBoundEvaluationPolicy`. A class modeling this concept must contain a method to get a separation bound for a given node and several methods to update the necessary parameters at certain occasions, for example when an operator node is converted to a bigfloat during the evaluation. The separation bound is expected to be retrieved by computing an algebraic degree bound for the node and calling the respective method of the given `SeparationBound` class. Four different strategies for computing the algebraic degree are implemented, as shown in purple in Figure 3.1. Although only one of these strategies is used for the experiments in this work, we briefly examine all four methods. The `Basic_sepd_evaluation` class implements the default behavior of `Real_algebraic`, i.e., it computes an algebraic degree bound using a depth-first traversal of the respective subexpression whenever a separation bound is requested. The class `Timestamped_sepd_evaluation` is similar to the previous one with the exception that once computed bounds are cached with a timestamp. If a separation bound is requested multiple times for the same node, the cached value can be returned as long as there have not been any changes to the expression dag that might have changed the degree bound.

For the `Timestamped_fully_cached_evaluation`, a bottom-up computation scheme for the algebraic degree bound is established. While the number of common subexpressions in an expression dag might be large, we can expect the dag to contain only a small number of *radical nodes*, i.e., nodes representing a root operation. Otherwise, the resulting separation bound is too small to achieve a feasible running time anyway. Building on this observation, for each node v all radical nodes in the subexpression rooted at v can be stored in a list at v. If the number of radical nodes is bounded by a small constant, storing them at each node does not have a significant impact on the running time. Now the list of radical nodes at v can be acquired from its children by merging their lists of radical nodes while removing duplicates and adding v itself if it contains a root operation. The list of radical nodes can be further reduced by listing only those nodes who have more than one parent as determined by their reference counter. In this case, each node additionally needs to store a degree factor that is unique to their subexpression. Consequently, in this case, all nodes with a unique degree larger than one must be treated like radical nodes.

With the bottom-up, fully-cached strategy, each node is traversed at most once as long as there are no changes to the expression dag. A degree bound can easily be computed for each node by multiplying the degrees of the listed nodes with the unique degree of the node itself. If the expression dag changes in a way that influences the degree bound, the stored parameters at each node must be recomputed. This can be done in two ways. With a `Strict_recursion`, similar to the previous strategies, all descendants are traversed and recomputed. The class `Faithful_recursion` on the other hand expects that all child nodes have already been recomputed if the data is invalidated and therefore performs only a local recomputation of its parameters, using the parameters at its direct children. The second strategy can only be used in combination with a `Topological_evaluation`. In contrast to the other strategies, however, it is guaranteed to be multithread-safe.

The final strategy, called `Faithful_fully_cached_evaluation`, is the one used in this work. It behaves similar to the previous strategy with faithful recursion, with the exception that it does not store a timestamp and therefore data is never globally invalidated. Instead, it relies on a computation order that will never require the cache to be reset. As `Faithful_recursion`, this class is multithread-safe, but can only be used with `Topological_evaluation`. A faithful evaluation is required for the experiments in Chapter 5. For consistency, we use the class `Faithful_fully_cached_evaluation` in all experiments in this work.

3.2.4 Error Handling

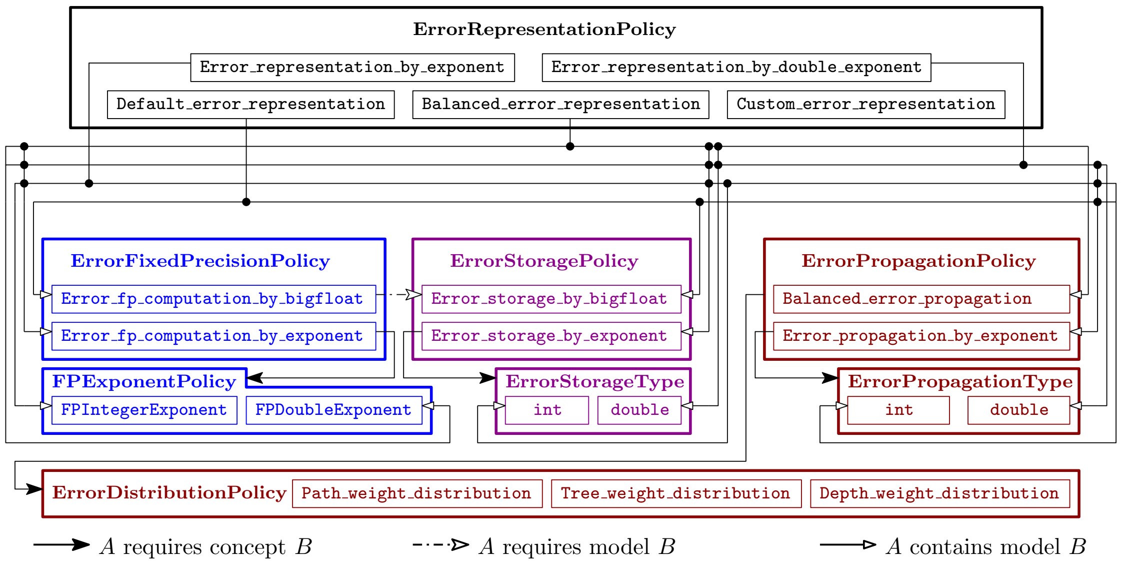

ErrorRepresentationPolicy
Error_representation_by_exponent
Error_representation_by_double_exponent
Default_error_representation
Balanced_error_representation
Custom_error_representation
ErrorFixedPrecisionPolicy
Error_fp_computation_by_bigfloat
Error_fp_computation_by_exponent
ErrorStoragePolicy
Error_storage_by_bigfloat
Error_storage_by_exponent
ErrorPropagationPolicy
Balanced_error_propagation
Error_propagation_by_exponent
FPExponentPolicy
FPIntegerExponent
FPDoubleExponent
ErrorStorageType
int
double
ErrorPropagationType
int
double
ErrorDistributionPolicy
Path_weight_distribution
Tree_weight_distribution
Depth_weight_distribution
A requires concept B
A requires model B
A contains model B

Figure 3.2: The concepts used for models of the `ErrorRepresentationPolicy`. The policy is subdivided into fixed precision computation (blue), error storage (purple) and accuracy-driven propagation (dark red). Each model of the error representation is a collection of models of sub-concepts as indicated by the respective top-down lines. The custom error representation allows the user to choose the models for the sub-concepts manually (not indicated in the figure).

During the evaluation process, error bounds for the current approximation are maintained. The `ErrorRepresentationPolicy` defines which representation is used for the error bounds and how error bounds are computed at different stages throughout the evaluation. The policy divides into three main aspects, namely the storage, the initial error bound computation at fixed precision and the error propagation during the accuracy-driven evaluation. In each of the three stages, the representation used can be chosen independently, with the exception of a bigfloat fixed-precision computation, which requires a bigfloat storage type. Choosing a representation has various consequences on the methods used for determining error bounds and, hence, their size. Different methods for the accuracy propagation play a main role in Chapter 4. The class `Balanced_error_propagation` reflects the strategies introduced in Section 4.2. The different types of representations for error bounds are explained in more detail in Chapter 6.

In the `ErrorRepresentationPolicy` concept, the three different aspects of error representation are subsumed in order to make error representation more accessable. Four different classes are defined to be used as a comprehensive error representation policy. The policy combinations for each of the four classes is shown in Figure 3.2. The default representation and the exponent representations can be used directly without further considerations. If the class `Balanced_error_representation` is used, additionally defining an `ErrorDistributionPolicy` is required (cf. Figure 3.1). A converter template, not depicted in any of the figures, is used to convert any error representation into any of the others. The class `CustomErrorRepresentation` may be used if the three subpolicies should be assigned manually by the user.

4 Optimizing Expression DAGs for Large-Scale Computations

Exact number types are currently designed to efficiently handle small computations. This is justified by the fact that in many geometric algorithms the numerical computation is restricted to the evaluation of small predicates, such as the orientation predicate or the incircle predicate. Nevertheless, there are several reasons why it is worthwhile to consider large-scale computations as well:

1. The data on which geometric algorithms operate is usually expected to be exact or at least not to be contradictory. This may not always be the case if the data itself is generated by an algorithm and the output is rounded to primitive number types. A loss of exactness can be avoided if the data itself is saved as an exact number type. In this case, expression dags can grow considerably over several iterations.

2. Some geometric algorithms process the same data over many iteration steps where the newly generated data is computed from the previous data and therefore, similar to before, the size of the involved expressions increases with each iteration.

3. The usage of exact computation in other algorithmic contexts is currently inhibited by the low performance of exact number types. Increasing the performance for large computations is a step toward a more general usage of exact number types, for example in the context of automated reasoning.

In the first section of this chapter, we discuss the cost associated with the evaluation of an expression dag and show how it is affected by a bad structure. In particular, we show that an unbalanced graph structure leads to a substantial increase in evaluation cost. Unbalanced structures occur naturally in algorithms. Whenever a simple loop construct is used, this usually leads to an unbalanced expression. While in theory the programmer can adjust their implementation, building a balanced expression demands considerable effort and usually requires to save intermediate results. In an online setting, the full expression might not even be known in advance. It is natural to rely on mechanisms provided by a general purpose number type instead. Moreover, an exact-decisions number type already maintains the computation history and is therefore well-suited to tackle these problems. In Section 4.2, we show how the effects of an unbalanced expression dag can be mitigated by adjusting the evaluation process to fit its structure. In Section 4.3, we discuss in what situations the overall structure of the graph can be restructured by the number type before an evaluation is started. In Section 4.4, we provide experiments to validate and to evaluate the theoretical results.

4.1 The Cost of Evaluating Expression DAGs

The concept of an expression dag, as described in Section 2.2.4, is central to exact number types based on accuracy-driven evaluation. If floating-point filters or similar methods fail to determine the sign of an expression, high-precision approximations are computed for each node in an expression dag to get a more accurate value for the expression. We refer to this process as the evaluation of the expression dag. In this section, we develop a cost model that is used in the subsequent analysis of the evaluation process. Expression dags in this work are always expected to operate on floating-point numbers and to support all radical expressions, i.e., all operators in $\Omega = \{\overset{u}{-}, +, -, \cdot, /\} \cup \{\sqrt[d]{\bullet} \mid d \in \mathbb{N}\}$ (cf. Section 2.2.5). Furthermore, our notion of accuracy always refers to an absolute error bound if not specified otherwise. We start with the definition of some basic concepts.

$\mathbf{V}(E)$, $\mathbf{E}(E)$

$\mathbf{V}_0(E)$, $\mathbf{E}_0(E)$

Definition 4.1. Let E be an expression dag. Then we denote the set of operator nodes of E as $\mathbf{V}(E)$ and the set of edges leading to an operator node as $\mathbf{E}(E)$. We furthermore denote the set of all nodes of E as $\mathbf{V}_0(E)$ and the set of all edges of E as $\mathbf{E}_0(E)$.

requested accuracy

required accuracy

operator precision

When an expression dag is evaluated accuracy-driven, a *requested accuracy* is associated with each of its nodes (cf. Section 2.2.6). In order to guarantee the requested accuracy, at each operator node *required accuracies* for the associated operation and for each of its edges are computed. The required accuracy for the operation is then translated into the node's *operator precision* and the required accuracies for the edges lead to a requested accuracy at their target. These concepts are formalized by the definition of an error distribution.

Definition 4.2. Let E be an expression dag. A (partial) function

$$q : \{E\} \cup \mathbf{V}(E) \cup \mathbf{E}(E) \to \mathbb{R}$$

error distribution

is called a *(partial) error distribution* for E.

$q(v)$

$q(e)$

$q(E)$

Semantically, an error distribution formalizes the notion of required accuracy for the elements of an expression dag. In the context of an error distribution, we generally represent accuracies by their magnitude as this representation more naturally transfers to a meaningful cost model. Consequently, for an expression dag E with error distribution q, we say that the required accuracy for a node $v \in \mathbf{V}(E)$ is $q(v)$ if we require the absolute operation error at v to be less or equal to $2^{q(v)}$. Similarly, we say the required accuracy for an edge $e = (u, v) \in \mathbf{E}(E)$ is $q(e)$ if we require the error of the approximation of the subexpression at v to be less or equal to $2^{q(e)}$. Finally, we say that an accuracy of $q(E)$ is required for E if the absolute error of the approximation for the value of E is required to be less or equal to $2^{q(E)}$.

A partial error distribution, and therefore the notion of required accuracies, can naturally be extended to all nodes in an expression dag.

Definition 4.3. Let E be an expression dag and let q be a partial error distribution for E. Then we call the function

$$\hat{q} : \{E\} \cup \mathbf{V}_0(E) \cup \mathbf{E}_0(E) \to \mathbb{R} \cup \{-\infty, \infty\}$$

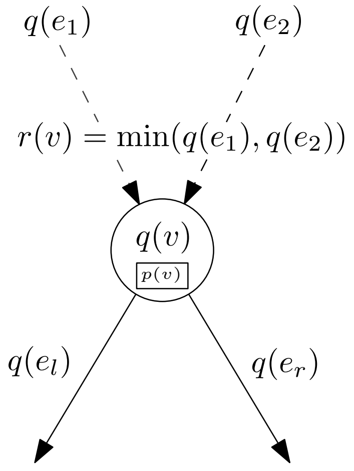


Figure 4.1: The basic accuracy propagation process. The requested accuracy $r(v)$ at an operator node v is determined by the required accuracies of the two incoming edges $e_1 = (u_1, v), e_2 = (u_2, v)$. The requested accuracy is then split into the required accuracy at the node and the required accuracies at the two outgoing edges e_l, e_r. The precision for the operation at v is derived from the required accuracy at v.

with

$$\hat{q}(x) = \begin{cases} q(x), & \text{if } q \text{ is defined for } x \\ -\infty, & \text{if } x \in (\mathbf{V}_0(E) - \mathbf{V}(E)) \cup (\mathbf{E}_0(E) - \mathbf{E}(E)) \\ \infty, & \text{otherwise} \end{cases}$$

the *natural extension* of q. natural extension

A leaf in an expression dag is always represented exactly and therefore infinitely accurate. If an error distribution does not specify the required accuracy of a non-leaf element of the expression dag, its value is not important for the respective evaluation step and therefore any accuracy is sufficient. In the following, we always assume a natural extension of an error distribution if a value would not be defined otherwise.

The requested accuracy at a node $v \in \mathbf{V}_0(E)$ represents the maximum accuracy necessary to fulfill all requirements of predecessors in the expression dag. It therefore depends on the required accuracies of all of its incoming edges. For an expression dag E with root v_0 and an error distribution q we set

$$r(v) = \begin{cases} q(E), & \text{if } v = v_0 \\ \min_{(u,v)\in\mathbf{E}_0(E)} q((u,v)), & \text{otherwise} \end{cases} \tag{4.1}$$ $r(v)$

and use $r(v)$ to denote the requested accuracy at v. We call $r(v_0) = q(E)$ the *target accuracy* of the evaluation. Figure 4.1 shows the basic accuracy propagation process. Both the computation of the required accuracies from the requested accuracy and the computation of the operator precision $p(v)$ from the accuracy required for the operation target accuracy are dependent on the node's operator. In the following section, we describe how these values are determined.

4.1.1 Accuracy Propagation

With a meaningful error distribution, the requested accuracy at a node $v \in \mathbf{V}(E)$ is guaranteed if the required accuracies at v and at its outgoing edges are guaranteed. In order to arrive at a definition for a meaningful error distribution, we have to specify how a requested error can be distributed among the operation error and the errors in the operands.

Notation 4.4. Let v be a node in an expression dag with value z. Then we denote the approximation for z stored at v by $\tilde{z}$. If v is an operator node with operator $\circ$, we denote the inexact operator that performs the operation at a certain operator precision by $\circledcirc$, i.e., we draw a circle around the operator.

$\tilde{z}$

$\circledcirc$

Let v be an operator node in an expression dag E with value z and operator $\circ$. Let σ be the absolute error caused by the operation at v. If $\circ$ is a unary operator, v has exactly one child. Let x be the value at the child node of v. Then the absolute error χ of the approximation $\tilde{z}$ is bounded by

$$\chi = |\tilde{z} - z| = |\circledcirc(\tilde{x}) - \circ(x)| \leq \sigma + |\circ(\tilde{x}) - \circ(x)|$$

Likewise, if $\circ$ is binary, then v has two children with values x, y and the absolute error of $\tilde{z}$ is bounded by

$$\chi = |\tilde{z} - z| = |\tilde{x} \circledcirc \tilde{y} - x \circ y| \leq \sigma + |\tilde{x} \circ \tilde{y} - x \circ y|$$

Thereby, the operation error can always be isolated from the errors in the operands. For some operators, there are error bound models which are able to capitalize on the operation error by not separating it from the error caused by the child approximations. In order to obtain a standardized format, we do not follow these approaches. Instead we determine so-called *operator constants* c_l, c_r for each operator $\circ \in \Omega$ such that we get an inequality of the form

operator constant

$$\chi \leq \sigma + c_l\varepsilon + c_r\delta \tag{4.2}$$

where ε and δ denote the errors at the left and the right child. For unary nodes, we have only a left child and therefore set $c_r = \delta = 0$. While the operator constants are not necessarily actual constants, they can be considered constant with respect to some aspects of the evaluation. We elaborate on this observation in Section 4.1.2. For the operator node itself, the error is bounded by the precision at which the operation is performed. We determine a *correction addend* $\gamma(v)$ such that the operator precision at v is given as

correction addend

$p(v)$

$$p(v) = -q(v) + \gamma(v) \tag{4.3}$$

For a correctly rounded operation, the operator precision can be directly translated into an upper bound to the relative operation error. If we guarantee $\gamma(v) \geq \log(|\tilde{z}|)$, then the absolute operation error is bounded by

$$\sigma \leq |\tilde{z}| \cdot 2^{-p(v)} \leq |\tilde{z}| \cdot 2^{q(v)} \cdot 2^{-\log(|\tilde{z}|)} \leq 2^{q(v)} \tag{4.4}$$

Note that the actual precision applied to an operation must necessarily be positive and integral. If the operator precision is negative or, in an actual implementation, smaller than the precision of a primitive operation, then the operation is evaluated at a fixed minimum precision. If the value of the operator precision is not integral, the rounded up precision is used for the actual computation.

Negation

For a negation only the sign bit of the approximation is changed. Therefore, negations do not affect the accuracy of the approximation as long as the resulting bigfloat can hold the same number of bits as the approximation of the child node. So we have $\chi = \varepsilon$ and take over the precision from the child operation. Throughout this work, we usually ignore negations in the analysis since they neither affect the accuracy needed in the subexpression nor have a significant effect on the running time.

Addition and Subtraction

Additions and subtractions behave similar in terms of error analysis. Since a subtraction is equivalent to an addition with a negated second operand, we generally do not distinguish between these two operations. In both cases, absolute errors in the operands simply add up to an absolute error in the result.

$$\begin{aligned}
\chi &\leq \sigma + |\tilde{x} \pm \tilde{y} - (x \pm y)| \\
&\leq \sigma + |(x+\varepsilon) \pm (y \pm \delta) - (x \pm y)| \\
&= \sigma + |(x \pm y) + \varepsilon + \delta - (x \pm y)| \\
&\leq \sigma + \varepsilon + \delta
\end{aligned}$$

For additions, the size of the resulting approximation is at most twice the size of its operands. With (4.4) it is therefore sufficient to set the operator precision to

$$p(v) = -q(v) + \max(\log(|\tilde{x}|), \log(|\tilde{y}|)) + 1 \geq -q(v) + \log(|\tilde{z}|)$$

Multiplication

In multiplication nodes, an error in the child node approximations is magnified by the size of the factors. To bound this error, we must obtain an upper bound on the size of x. Let $x_{high} = |\tilde{x}| + \varepsilon$. Then $|x| \leq x_{high}$ and we get

$$\begin{aligned}
\chi &\leq \sigma + |\tilde{x} \cdot \tilde{y} - x \cdot y| \\
&\leq \sigma + |\,((x+\varepsilon) \cdot \tilde{y}) - (x \cdot y)\,| \\
&= \sigma + |x \cdot \tilde{y} + \varepsilon \tilde{y} - (x \cdot y)| \\
&\leq \sigma + |x \cdot (y+\delta) + \varepsilon \tilde{y} - (x \cdot y)| \\
&= \sigma + |x \cdot y + \varepsilon \tilde{y} + x\delta - (x \cdot y)| \\
&\leq \sigma + |\tilde{y}|\varepsilon + x_{high}\delta
\end{aligned}$$

Note that the resulting term seems surprisingly asymmetric. The reason for this asymmetry is the incorporation of an error of size $\varepsilon\delta$ caused by the multiplication of the error terms. With a similar argument we can bound χ by $\sigma + y_{high}\varepsilon + |\tilde{x}|\delta$ with $y_{high} = |\tilde{y}| + \delta$. In `Real_algebraic`, a symmetry is established by using both x_{high} and y_{high} in order to bound $|x|$ and $|\tilde{y}|$, respectively. While certainly not the optimal strategy, we do not change this behavior for this work. The magnitude of the approximation after a multiplication is bounded by the product of the magnitudes of the operands. Therefore, the required accuracy at a node v translates into the operator precision as

$$p(v) = -q(v) + \log(|\tilde{x}|) + \log(|\tilde{y}|) \geq -q(v) + \log(|\tilde{z}|)$$

and the operation error is bounded as in (4.4). Note that the magnitude of the approximation is always smaller or equal to the product of the magnitudes of the operands if the operation is rounded toward zero. If the magnitudes are rounded to the next largest integer, the validity of the statement does not depend on the rounding mode of the operation anymore.

Division

Let $x_{high} = |\tilde{x}| + \varepsilon$ and $y_{low} = |\tilde{y}| - \delta$ and assume $y_{low} > 0$. Then the error at a division node is bounded by

$$\begin{aligned}
\chi &\leq \sigma + \left|\frac{\tilde{x}}{\tilde{y}} - \frac{x}{y}\right| \\
&\leq \sigma + \left|\frac{(x+\varepsilon)y - x(y-\delta)}{\tilde{y}y}\right| \\
&= \sigma + \left|\frac{1}{\tilde{y}}\varepsilon + \frac{x}{\tilde{y}y}\delta\right| \\
&\leq \sigma + \frac{1}{|\tilde{y}|}\varepsilon + \frac{x_{high}}{|\tilde{y}|y_{low}}\delta
\end{aligned}$$

Since $|\tilde{y}| \geq y_{low}$, we can always use y_{low} in the denominator, which may save one bigfloat operation, depending on the implementation. Note that for computing this bound it is not sufficient to guarantee $y > 0$, but instead we must make sure that $y_{low} > 0$ before starting the evaluation. Similarly to multiplication, the magnitude of the approximation is bounded by the product of the magnitude of the numerator and the magnitude of the inverted denominator. We set the operator precision to

$$p(v) = -q(v) + \log(|\tilde{x}|) - \log(|\tilde{y}|)$$

and get a bound to the absolute operation error as in (4.4).

Root of Degree d

Let $x_{low} = |\tilde{x}| - \varepsilon$ and assume $x_{low} > 0$. For a function f with monotonically decreasing slope, we can bound the value of $f(a+b)$ from above by $f(a) + b\frac{d}{dt}f(t)|_a$. The error for a root operation of degree d is therefore bounded by

$$\begin{aligned}\chi &\leq \sigma + |\sqrt[d]{\tilde{x}} - \sqrt[d]{x}| \\ &\leq \sigma + |\sqrt[d]{x+\varepsilon} - \sqrt[d]{x}| \\ &\leq \sigma + |\sqrt[d]{x} + \varepsilon\tfrac{d}{dt}\sqrt[d]{t}\,|_x - \sqrt[d]{x}| \\ &= \sigma + |\tfrac{1}{d}x^{\frac{1}{d}-1}\varepsilon| \\ &\leq \sigma + \tfrac{1}{d}x_{low}^{\frac{1-d}{d}}\varepsilon\end{aligned}$$

For determining the operator precision, an upper bound on the magnitude of the resulting approximation must be computed. When applying the logarithm, computing the d-th root becomes a division by d. The operator precision is set to

$$p(v) = -q(v) + \log(|\tilde{x}|)/d$$

and a bound to the operation error follows with (4.4).

Summary

The operator constants are summarized in Table 4.2. The occurrences of the value of the approximation have been replaced by the upper or lower bounds of the intervals in order to simplify the equations. Note that we can assign the operator constants at each node to its outgoing edges by assigning c_l to the left outgoing edge and, if the node is binary, c_r

	negation	add./sub.	multipl.	division	d-th root
c_l	1	1	y_{high}	$\frac{1}{y_{low}}$	$\frac{1}{d}(x_{low})^{\frac{1-d}{d}}$
c_r	0	1	x_{high}	$\frac{x_{high}}{y_{low}^2}$	0

Table 4.2: Operation-dependent constants c_l and c_r for an accuracy-driven evaluation in `Real_algebraic` with x_{high}, y_{high} upper bounds and x_{low}, y_{low} lower bounds on the child values.

to the right outgoing edge. For an expression dag E and an edge $e \in \mathbf{E}_0(E)$, we denote the operation constant along this edge as $c(e)$. The correction addends are displayed in Table 4.3. For negations, the operation error is always zero if the precision is set to the precision of the child node. Hence, it is not determined by the required accuracy of the node and the concept of correction addends is not applicable. Note that, in contrast to the operator constants, the correction addends are defined in a logarithmic context.

$c(e)$

	$\gamma(v)$
add./sub.	$\max(\log(\lvert\tilde{x}\rvert), \log(\lvert\tilde{y}\rvert)) + 1$
multiplication	$\log(\lvert\tilde{x}\rvert) + \log(\lvert\tilde{y}\rvert)$
division	$\log(\lvert\tilde{x}\rvert) - \log(\lvert\tilde{y}\rvert)$
d-th root	$\log(\lvert\tilde{x}\rvert)/d$

Table 4.3: Operation-dependent correction addend $\gamma(v)$ for an accuracy-driven evaluation in `Real_algebraic`.

4.1.2 The Cost Function

In this section, we develop a simple but meaningful model for the "cost" of an accuracy-driven evaluation. Empirically, most of the running time of an exact number type such as `Real_algebraic` is reserved for the underlying bigfloat arithmetic unless the computation can be avoided through early filter mechanisms. As long as other algorithms, that are executed during the evaluation and use only primitive types, are reasonably fast (linear up to small-constant quadratic running time), they can be expected to have little impact on the overall running time. For this reason, we choose to focus exclusively on the cost caused by bigfloat operations. Let p be the precision at which a bigfloat operation should be executed. As described in Section 2.1, all standard bigfloat operations can be done in $O(p \log p)$. In efficient bigfloat libraries, such as `MPFR`, all operations are implemented with algorithms that guarantee a running time of $O(p(\log p)^k)$ for a small $k \in \mathbb{N}$. We therefore can expect an *almost linear behavior*, i.e., if we execute n operations with precision p, the running time is $nT(p) \sim T(np)$.

almost linear

During one evaluation of an expression dag, the error distribution may change. We assign an *evaluation event* to each bigfloat operation that is executed during the evaluation process. We denote the set of all evaluation events occurring during a certain evaluation of an expression dag E by $H(E)$. For an evaluation event $h \in H(E)$ we denote the error distribution at this event by q_h. Likewise, we denote the requested accuracy at a node $v \in \mathbf{V}(E)$ during h by $r_h(v)$ and the operator precision at this node by $p_h(v)$. We can now define the cost of evaluating an expression dag as the sum of the precision needed for all bigfloat operations.

evaluation event

$H(E)$

q_h

$r_h(v)$, $p_h(v)$

Definition 4.5. Let $H(E)$ be the set of bigfloat evaluation events occurring during one accuracy-driven evaluation of an expression dag E. Let $h \in H(E)$ be a single evaluation event and let $v \in \mathbf{V}(E)$ be the node that is evaluated during h. Then the *true cost of h* is defined as $\operatorname{cost}_t(h) = p_h(v)$. Furthermore, we define the *true cost of E* as $\operatorname{cost}_t(E) = \sum_{h \in H(E)} \operatorname{cost}_t(h)$. true cost

In the definition of the true cost, we do not take performance differences between operations into account. While in practice there are significant differences in the constants associated with the different algorithms (cf. Table 2.3, page 16), we choose to ignore them in the cost model in order to make expression dags with mixed operations analyzable. Since the true cost is based on the precision of the operations, it evolves around the relative accuracies present in the subexpressions. In the context of an accuracy-driven evaluation, however, we mostly deal with absolute instead of relative accuracies. For the analysis, it is therefore more convenient to have a model that is based on absolute accuracy.

Definition 4.6. Let $H(E)$ be the set of bigfloat evaluation events occurring during one accuracy-driven evaluation of an expression dag E and let $h \in H(E)$ be an evaluation event occurring at $v \in \mathbf{V}(E)$. Then the *absolute cost of h* is defined as $\operatorname{cost}(h) = -q_h(v)$. Furthermore, we define the *absolute cost of E* as $\operatorname{cost}(E) = \sum_{h \in H(E)} \operatorname{cost}(h)$. absolute cost

In the previous section, it was shown how the operator precision can be computed from the absolute accuracy required for the operation. Let $\gamma(h)$ be the correction addend at the evaluated node v as described in Table 4.3. Then the true cost of an evaluation is equal to the sum of the absolute cost of the evaluation and the addends, i.e.,

$$\operatorname{cost}_t(E) = \operatorname{cost}(E) + \sum_{h \in H(E)} \gamma(h)$$

The correction addends depend on the sizes of the respective approximations. These sizes can vary heavily depending on the actual values in the leaves and the structure of the expression dag. Knowing these sizes in advance is in general equivalent to the knowledge about the sign of the (sub-)expression. At the same time, the choices we make during the evaluation usually do not lead to a drastic change in their magnitude. If applied carefully, the absolute cost is therefore a good indicator for the behavior of the true cost. In the following, we always refer to the absolute cost if not specified otherwise.

The requested accuracy at a node and, consequently, the required accuracy for its operation depend on the required accuracies from its parent nodes (cf. Equation 4.1). Since the same applies to the parents, the required accuracy at a node depends on all of its predecessors in the expression dag. The concept of a root path is therefore fundamental for the analysis of the evaluation cost.

Definition 4.7. Let E be an expression dag. For $k \in \mathbb{N}_0$ let $\{v_0, ..., v_k, v\} \subseteq \mathbf{V}_0(E)$ be a set of nodes in E and let $\{e_0, ..., e_k\} \subseteq \mathbf{E}_0(E)$ be a set of edges in E, such that $e_i = (v_i, v_{i+1})$ for $0 \leq i < k$ and $e_k = (v_k, v)$. Then we call the $(2k+1)$-tuple

$$P = (v_0, e_0, ..., v_k, e_k, v)$$

a *path between v_0 and v in E*. If v_0 is the root of E, then we call P a *root path* of v. We path, root path

$\mathcal{P}(v)$ $\mathbf{V}(P)$, $\mathbf{E}(P)$ denote the set of root paths for a node v by $\mathcal{P}(v)$. As for expression dags, we denote the set of nodes in P as $\mathbf{V}(P)$ and the set of edges along P as $\mathbf{E}(P)$.

In an error distribution, at each edge along a path in E the required accuracy may change. The following definition formalizes the total change in accuracy along a path in E.

$i(e)$ $i(v)$ **Definition 4.8.** Let $e = (u, v) \in \mathbf{E}_0(E)$ be an edge in an expression dag E with error distribution q. We call $i(e) = q(e) - r(u)$ the *accuracy increase along e*. For a node $v \in \mathbf{V}_0(E)$ we call $i(v) = q(v) - r(v)$ the *accuracy increase at v*. Finally, for a path $P = (v_0, e_0, ..., e_k, v_{k+1})$ we call

$i(P)$
$$i(P) = \sum_{i=0}^{k} i(e_i) + i(v_{k+1})$$

the *accuracy increase along P*.

The requested accuracy at a node is defined by the maximum accuracy required along all of its root paths. So, naturally, some paths in an expression dag are especially relevant for the evaluation.

defining path **Definition 4.9.** Let E be an expression dag with error distribution q, let $u, v \in \mathbf{V}(E)$ and let P be a path between u and v in E. Then the path P is called a *defining path* if $i(P) = q(v) - r(u)$.

Observation 4.10. Let E be an expression dag with error distribution q whose value should be determined with target accuracy $q(E) = z$. If for a node $v \in \mathbf{V}(E)$ a root path $P \in \mathcal{P}(v)$ is defining, the required accuracy for the node's operation is given by

$$q(v) = z + i(P)$$

Remark 4.11. Throughout this work, we use z to denote the absolute accuracy that should be guaranteed for the final approximation by the accuracy-driven evaluation. Since in a reasonable error distribution $q(E) = z$, we use the term *target accuracy* for this value as well (cf. page 45).

If a path P is defining, then all subpaths of P must be defining as well. In particular, the edges along the path are the edges that define the requested accuracy of their target nodes. Conversely, a path consisting of these edges is always defining.

Lemma 4.12. Let $P = (v_0, e_0, ..., e_k, v_{k+1})$ be a path in an expression dag with error distribution q. Then P is defining if and only if $q(e_i) = r(v_{i+1})$ for all $0 \le i \le k$.

Proof. If $q(e_i) = r(v_{i+1})$ for all edges then

$$i(P) = \sum_{i=0}^{k} i(e_i) + i(v_{k+1})$$

$$
\begin{aligned}
&= \sum_{i=0}^{k} (q(e_i) - r(v_i)) + q(v_{k+1}) - r(v_{k+1}) \\
&= \sum_{i=0}^{k} (r(v_{i+1}) - r(v_i)) + q(v_{k+1}) - r(v_{k+1}) \\
&= q(v_{k+1}) - r(v_0)
\end{aligned}
$$

and therefore P is defining. If, on the other hand, P is defining, then we must have

$$\sum_{i=0}^{k} q(e_i) = \sum_{i=0}^{k} r(v_{i+1})$$

Since the requested accuracy at a node is the highest of the required accuracies of all incoming edges (cf. Equation 4.1), we have $q(e_i) \geq r(v_{i+1})$ for all i and therefore $q(e_i) = r(v_{i+1})$ for all edges. □

Corollary 4.13. Let E be an expression dag with an error distribution q. Then for each $v \in \mathbf{V}(E)$ at least one path in $\mathcal{P}(v)$ is defining.

Proof. The requested accuracy at a node is the minimum of the required accuracies of all, finitely many, incoming edges. For each node v' in the expression dag there is at least one incoming edge e, such that $r(v') = q(e)$. Let P be a root path of v created by traveling toward the root using only such edges. Then, with Lemma 4.12, P is defining. □

During a recursive evaluation, whenever an accuracy is required at a node that is higher than the current accuracy of the approximation, the bigfloat operation at the node is re-evaluated. In the worst case, for each node $v \in \mathbf{V}(E)$ each path $P \in \mathcal{P}(v)$ requires a different accuracy and each of those paths becomes defining at one point of the algorithm (cf. [MS15]). Then the total cost of an evaluation to accuracy z is given as

$$\operatorname{cost}(E) = -\sum_{v \in \mathbf{V}(E)} \sum_{P \in \mathcal{P}(v)} (z + i(P))$$

In a topological evaluation the highest requested accuracy for each node is computed before any evaluation event occurs. Afterwards, each node is evaluated exactly once, assuming that there are no pre-evaluated nodes. Therefore, the total cost is guaranteed to be

$$\operatorname{cost}(E) = -\sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} (z + i(P)) = -nz - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} i(P) \tag{4.5}$$

where n denotes the number of operator nodes in E. For the rest of this work, we always assume topological evaluation if not explicitly stated otherwise. We generally assume that, in any expression dag, no node has been evaluated before. By using a topological approach we can then refrain from using evaluation events to define costs. Instead, we assign a cost to each operator node.

Definition 4.14. Let E be an expression dag with error distribution q. For an operator node $v \in \mathbf{V}(E)$ we set the *true cost of* v to $\mathrm{cost}_{\mathrm{t}}(v) = p(v)$ and the *absolute cost of* v to $\mathrm{cost}(v) = -q(v)$.

$\mathrm{cost}_{\mathrm{t}}(v)$
$\mathrm{cost}(v)$

Now, the total cost of an evaluation can be represented directly as the sum of the cost of all operator nodes.

Observation 4.15. For a topological evaluation without pre-evaluated nodes, we have $\mathrm{cost}_{\mathrm{t}}(E) = \sum_{v \in \mathbf{V}(E)} \mathrm{cost}_{\mathrm{t}}(v)$ and $\mathrm{cost}(E) = \sum_{v \in \mathbf{V}(E)} \mathrm{cost}(v)$.

With the observation and (4.5), we get a relation between the cost of a node and the cost increase along the paths in an expression dag.

Corollary 4.16. $\sum_{v \in \mathbf{V}(E)} \mathrm{cost}(v) = -nz - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} i(P)$ □

4.1.3 Cost and Structure

In Section 4.1.1, we described how the total error χ at a node v is bounded by the operation error and the errors at the child nodes. Let $\rho = 2^{-q(v)}$ be the error requested at a node v. We know how to compute constants c_l, c_r, such that

$$\chi \leq \sigma + c_l \varepsilon + c_r \delta$$

where σ is the operation error and ε, δ are the errors at the left and the right child (cf. Section 4.1.1). If the right side is bounded by ρ, the requested accuracy is guaranteed. For distributing the requested error among the three error terms at the right hand side of the inequality, it is important to note when one or both of the child nodes are exact.

Definition 4.17. For an expression dag E, an operator node $v \in \mathbf{V}(E)$ is called a *quasi-leaf* if all of its children are leaves. Furthermore, we call v *quasi-unary* if it has exactly one child node that is not a leaf and *fully binary* if it is binary and both children are operator nodes.

quasi-leaf
quasi-unary
fully binary

Leaves are always exact and therefore do not introduce errors. If v is a quasi-leaf, requiring $\sigma \leq \rho$ is therefore sufficient to guarantee $\chi \leq \rho$. Otherwise, the error must be distributed among the operation and the children. `Real_algebraic` chooses the required error bounds such that

$$\sigma \leq \frac{1}{2}\rho \quad \text{and} \quad c_l \varepsilon \leq \frac{1}{4}\rho \quad \text{and} \quad c_r \delta \leq \frac{1}{4}\rho \tag{4.6}$$

if v is fully binary and

$$\sigma \leq \frac{1}{2}\rho \quad \text{and} \quad c_l \varepsilon \leq \frac{1}{2}\rho \tag{4.7}$$

if v is quasi-unary. While this choice of an error distribution is arbitrary to some degree, using powers of two as fractions leads to an integral shift in the exponent. If accuracies are represented by their exponent, rounded to the next integer, this error distribution makes maximum use out of the inevitable rounding.

Definition 4.18. Let E be an expression dag and let z be the target accuracy to which E should be evaluated. Let q be the error distribution where $q(E) = z$, the required accuracy at a node $v \in \mathbf{V}(E)$ is computed from the requested accuracy as

$$q(v) = \begin{cases} r(v), & \text{if } v \text{ is a quasi-leaf} \\ r(v) - 1, & \text{else} \end{cases}$$

and the required accuracy at an edge $e = (u, v) \in \mathbf{E}(E)$ is computed as

$$q(e) = \begin{cases} r(u) - 1 - \lceil \log(c(e)) \rceil, & \text{if } u \text{ is quasi-unary} \\ r(u) - 2 - \lceil \log(c(e)) \rceil, & \text{if } u \text{ is fully binary} \end{cases}$$

Then we call q the *standard error distribution* (for E and z). standard error dist.

The standard error distribution makes the performance of the evaluation process vulnerable to bad structure. In particular, it leads to high cost if the expression dag has a great depth.

Definition 4.19. Let E be an expression dag and let $v \in \mathbf{V}_0(E)$. Then the *root distance* of v in E is the maximum number of edges along a path from the root of E to v, i.e., root distance

$$\text{dist}_E(v) = \max_{P \in \mathcal{P}(v)} |\mathbf{E}(P)|$$

$\text{dist}_E(v)$

The set of nodes with root distance j is called the *j-th level* of E. The *depth* of E is the maximum distance from the root to any of its leaves, i.e., level

$$\text{depth}(E) = \max_{v \in \mathbf{V}_0(E)} \text{dist}_E(v)$$

$\text{depth}(E)$

Furthermore, let E_v be the subexpression rooted at a node $v \in \mathbf{V}_0(E)$, then we call $\text{depth}(v) = \text{depth}(E_v)$ the *depth of v*. $\text{depth}(v)$

We call an expression dag an *(arithmetic) expression tree* if its operator nodes form a tree. Let E_{list} be a *list-like* expression dag with n binary operator nodes, i.e., an expression tree where the subgraph formed by the operator nodes is a linear list (cf. Figure 4.4). For simplicity, assume that all operations in E_{list} are additions and therefore for each edge e we have $c(e) = 1$. All operator nodes of E_{list} are quasi-unary, except for the node on the largest level, which is a quasi-leaf. Let $P(v)$ be the unique root path of a node $v \in \mathbf{V}(E_{list})$. Then the cost of evaluating E_{list} to accuracy z is expression tree list-like

$$\begin{aligned} \text{cost}(E_{list}) &= -nz - \sum_{v \in \mathbf{V}(E_{list})} \min_{P \in \mathcal{P}(v)} i(P) \\ &= -nz - \sum_{v \in \mathbf{V}(E_{list})} \left(\sum_{e \in \mathbf{E}(P(v))} i(e) + i(v) \right) \\ &= -nz + \sum_{i=0}^{n-2} (i+1) + (n-1) \end{aligned}$$

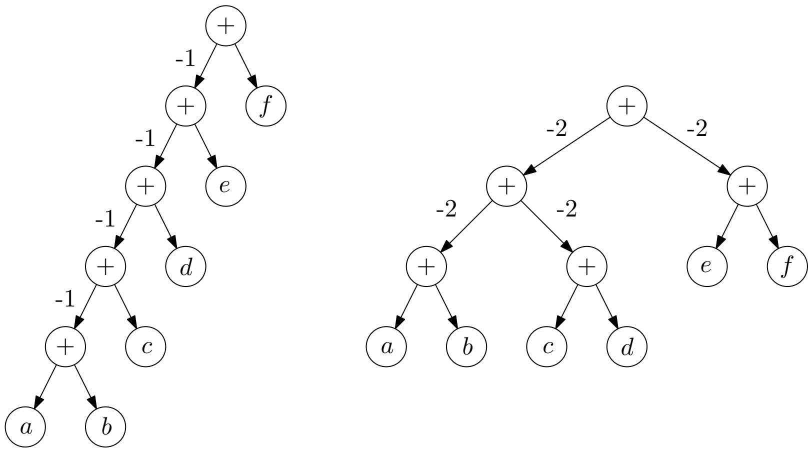


Figure 4.4: Depictions of a list-like (left) and a balanced (right) expression dag where all operations are additions. In the list-like expression dag the required accuracy of a node is on average affected by a linear number of accuracy increases. In the balanced case, nodes are only affected by a logarithmic number of accuracy increases.

$$= -nz + \frac{n(n+1)}{2} - 1$$

balanced We call an expression dag *balanced* if its operator nodes form a perfectly balanced tree. Let E_{bal} be a balanced expression dag with $n = 2^k - 1$ binary operator nodes. As before, assume that all operations in E_{bal} are additions and let $P(v)$ be the unique root path of a node $v \in \mathbf{V}(E_{bal})$. On the j-th level of E_{bal} there are 2^j nodes. Each operator node on the largest level is a quasi-leaf, while the other operator nodes are fully binary. The cost of evaluating E_{bal} to accuracy z is

$$\begin{aligned}
\operatorname{cost}(E_{bal}) &= -nz - \sum_{v \in \mathbf{V}(E_{bal})} \min_{P \in \mathcal{P}(v)} i(P) \\
&= -nz - \sum_{v \in \mathbf{V}(E_{bal})} \left(\sum_{e \in \mathbf{E}(P(v))} i(e) + i(v) \right) \\
&= -nz + \sum_{i=0}^{k-2} 2^i (2i+1) + 2^{k-1} 2(k-1) \\
&= -nz + 2 \sum_{i=0}^{k-1} 2^i i + \frac{n+1}{2} - 1 \\
&= -nz + 2(2^k(k-2)+2) + \frac{n+1}{2} - 1 \\
&= -nz + 2(n+1)\log(n+1) - 3.5n - 0.5
\end{aligned}$$

So the cost caused by the increases is of order n^2 for E_{list} and of order $n \log n$ for E_{bal}. Each increase in accuracy along an edge (u, v) leads to an increase in accuracy at all nodes in the subexpression rooted at v. Due to the list structure of E_{list}, such an increase affects on average $\Theta(n)$ nodes, whereas in the balanced case at most $O(\log n)$ nodes are affected. If operations other than additions are involved, this effect is magnified by the operator constants. In general, a more balanced structure leads to much lower cost than a structure containing long list-like subgraphs. In the following sections we discuss possible solutions to this problem.

4.2 Error Bound Balancing

In Section 4.1.3, a general connection between the structure of an expression dag and the cost of its evaluation is demonstrated. It is shown that the standard error distribution leads to an increased evaluation cost for unbalanced expressions. In this section, we show how to decrease the impact of structure on the evaluation cost by adaptively redistributing error bounds with respect to the graph structure. We call this method *error bound balancing* [GW19]. The options on how to distribute errors are limited by the representation of the error bound. In order to have significant impact on the evaluation cost, we must be able to define more precise error bounds than with an integer exponent representation. Since using bigfloats is very expensive, we choose to represent the accuracies propagated during the evaluation by their exponent using a primitive floating-point number type, namely a `double`. Advantages and limitations of this choice are discussed in Chapter 6. An error bound balancing strategy has not been implemented in any exact number type before this work. In 2006, Joris van der Hoeven theoretically described the concept of error bound balancing in the context of effective numbers [Hoe06a]. Although an implementation for effective numbers has been planned [Hoe06c], to the author's knowledge, it was never completed and/or published. In a paper by Yong Li and Jun-Hai Yong, heuristic balancing strategies for additions and multiplications are described, but without a thorough analysis and without the ambition of providing exact computation [LY07].

error bound bal.

In Section 4.2.1, we define a new error distribution based on a so-called *path weight*. We show various properties of this new distribution and, in the main theorem of this section, we show that this error distribution is uniquely optimal among all valid error distributions for an expression dag. In Section 4.2.2 we give a brief description of the implementation of the error distribution. In Section 4.2.3, we introduce two alternative weight functions which are in some cases easier to compute than the path weight. One of these heuristics is largely identical to the error distribution described by van der Hoeven.

4.2.1 The Path Weight Error Distribution

Let E be an expression dag with error distribution q and let $v \in \mathbf{V}(E)$ be a node in E with outgoing edges e_l, e_r. Let $\rho = 2^{r(v)}$ be the requested error bound at v and let $\sigma \leq 2^{q(v)}, \varepsilon \leq 2^{q(e_l)}$ and $\delta \leq 2^{q(e_r)}$ be the absolute errors that are expected with respect to the required accuracies at the operation, the left outgoing edge and the right outgoing edge of v. Finally, let c_l, c_r be the operation constants for v from Table 4.2. We want to choose the required accuracies such that the overall cost is minimized while satisfying

$$\begin{aligned} \sigma + c_l\varepsilon + c_r\delta \le \rho \quad &\Leftarrow \quad 2^{q(v)} + c_l 2^{q(e_l)} + c_r 2^{q(e_r)} \le 2^{r(v)} \\ &\Leftrightarrow \quad 2^{r(v)+i(v)} + c_l 2^{r(v)+i(e_l)} + c_r 2^{r(v)+i(e_r)} \le 2^{r(v)} \\ &\Leftrightarrow \quad 2^{i(v)} + c_l 2^{i(e_l)} + c_r 2^{i(e_r)} \le 1 \end{aligned} \tag{4.8}$$

balancing constraint

We call (4.8) the *balancing constraint*.

Remark 4.20. If either of the edges does not exist or leads to a leaf, the balancing constraint must be adjusted accordingly. In order to simplify the notation we sloppily assume that $c_r = 0$ if e_r does not exist (but $i(e_r)$ is defined) and that $i(e) = -\infty$ and therefore $2^{i(e)} = 0$ if e leads to a leaf.

If an error distribution always fulfills the balancing constraint, a computation according to this error distribution is guaranteed to yield correct error bounds. Using this property, we can specify when we consider an error distribution meaningful.

valid

Definition 4.21. Let E be an expression dag. An error distribution q for E is called *valid* if for each $v \in \mathbf{V}(E)$ it fulfills the balancing constraint.

With the derivation in Section 4.1.3, we have:

Observation 4.22. For each expression dag E and each target accuracy z, the standard error distribution for E and z is a valid error distribution.

In our further analysis, we assume that the size of the operator-dependent factors c_l, c_r is independent of the choice of the required accuracies, although in practice this may not be the case. If the value of a node is close to or exactly zero, a higher accuracy tendentially leads to a decrease in the size of the approximation. Furthermore, an approximation may be identified as exact which can lead to converting the whole subexpression to a single bigfloat node. These effects, however, are out of scope for our level of analysis and therefore ignored. Consequently, the correction addends for converting the operation accuracy to the operator precision are considered constant as well. We therefore can focus on the absolute cost of an evaluation and expect an optimal error distribution to be optimal with respect to true cost as well (cf. Section 4.1.2).

Notation 4.23. For an expression dag E and an error distribution q for E, we denote the (absolute) cost of E with respect to q as $\text{cost}(E, q)$ if the relation to q would otherwise not be obvious from the context.

optimal error distribution

Definition 4.24. Let E be an expression dag. Then a valid error distribution q in E is called *optimal* if $\text{cost}(E, q) \le \text{cost}(E, \hat{q})$ for all valid error distributions $\hat{q}$ in E with $q(E) = \hat{q}(E)$.

A path P in E influences the total cost only if it is a root path and P is defining. We first show that in an optimal error distribution, every path is a defining path.

Lemma 4.25. Let E be an expression dag with an optimal error distribution q and let $v \in \mathbf{V}(E)$ with outgoing edges e_l, e_r. Then $2^{i(v)} + c_l 2^{i(e_l)} + c_r 2^{i(e_r)} = 1$.

Proof. For each valid error distribution, the left side is less or equal to 1 due to the balancing constraint (4.8). Assume the left side is less than one. Then there is an $\varepsilon > 0$ such that we can increase $q(v)$ by ε without violating the constraint. It follows that with the new value for $q(v)$ the total cost

$$\text{cost}(E) = -\sum_{v' \in \mathbf{V}(E)} q(v')$$

decreases by ε, contradicting the optimality of the parameter choice. □

Lemma 4.26. Let E be an expression dag with an optimal error distribution and let P be a path in E leading to a node $v \in \mathbf{V}(E)$. Then P is defining.

Proof. Assume P is not defining. Then there exists an edge $e = (u, v) \in \mathbf{E}(P)$ where $q(e) > r(v)$. We can therefore decrease $q(e)$ by $q(e) - r(v) > 0$ without changing the cost of E, leading to another optimal error distribution. Since this implies a decrease in $i(e) = q(e) - r(u)$, the balancing constraint at u is not fulfilled by equality in the new error distribution. This is a contradiction to Lemma 4.25. Hence, P is defining. □

In order to find an optimal error distribution we associate a weight $w(e)$ with each edge $e \in \mathbf{E}_0(E)$. The weight can be loosely understood as the impact that a change in $q(e)$ has to the total cost of the expression dag. It is dependent on the operator constants along a path

Definition 4.27. Let E be an expression dag and let P be a path in E. We call

$$\text{cost}_\text{f}(P) = \sum_{e \in \mathbf{E}(P)} \log(c(e))$$

$\text{cost}_\text{f}(P)$

the *fixed cost* of P.

fixed cost

Definition 4.28. Let E be an expression dag. For $v \in \mathbf{V}_0(E)$ and $e \in \mathbf{E}_0(E)$ let

$$c_f(v) = \sum_{P \in \mathcal{P}(v)} 2^{\text{cost}_\text{f}(P)} \quad \text{and} \quad c_f(v, e) = \sum_{P \in \mathcal{P}(v) \,|\, e \in \mathbf{E}(P)} 2^{\text{cost}_\text{f}(P)}$$

$c_f(v)$, $c_f(v, e)$

Then the *path weight* of a node v is defined as

path weight

$$w(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ 1 + w(e_l) & \text{if } v \text{ is a unary operator with outgoing edge } e_l \\ 1 + w(e_l) + w(e_r) & \text{if } v \text{ is a binary operator with outgoing edges } e_l, e_r \end{cases}$$

$w(v)$

and the *path weight* of an edge $e = (u, v)$ is defined as

$$w(e) = \frac{c_f(v, e)}{c_f(v)} w(v) = \frac{\sum_{P \in \mathcal{P}(v) \,|\, e \in \mathbf{E}(P)} 2^{\text{cost}_\text{f}(P)}}{\sum_{P \in \mathcal{P}(v)} 2^{\text{cost}_\text{f}(P)}} w(v)$$

$w(e)$

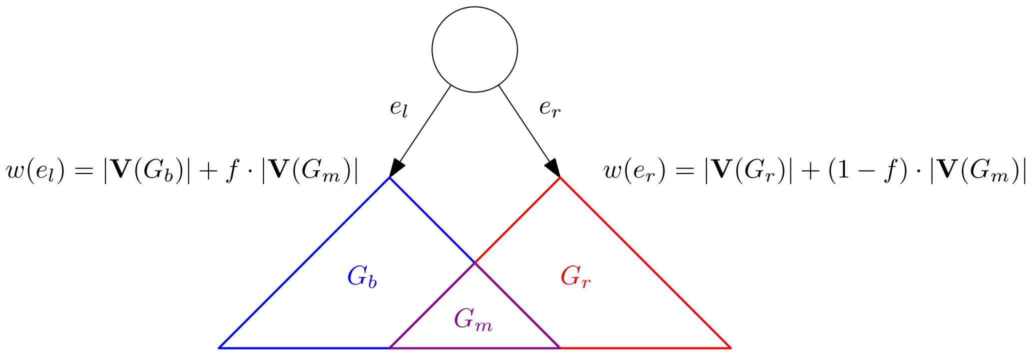

e_l
e_r
$w(e_l) = |\mathbf{V}(G_b)| + f \cdot |\mathbf{V}(G_m)|$
$w(e_r) = |\mathbf{V}(G_r)| + (1-f) \cdot |\mathbf{V}(G_m)|$
G_b
G_r
G_m

Figure 4.5: A simplified depiction of the path weight. The weight describes the number of operator nodes in the respective subexpression, whereas the weight of common subexpressions is shared among all parents. For a single common subexpression G_m with root v, this is described by a factor $f = \frac{c_f(v,e_l)}{c_f(v)}$.

The path weight basically counts the number of operator nodes in the subexpression rooted at the target node of an edge (cf. Figure 4.5). If a node possesses more than one root path, the path weight distributes the impact of the node among those paths according to their fixed costs. This property is expressed by the following lemma.

Lemma 4.29. For the path weight of an edge e in an expression dag E we have

$$w(e) = \sum_{v \in \mathbf{V}(E)} \frac{c_f(v,e)}{c_f(v)} \tag{4.9}$$

Proof. We prove this equivalence by induction. If $e = (v_0, v_1)$ leads to a leaf, then $w(e) = w(v_1) = 0$. Since no operator node is reachable through e, we have $c_f(v,e) = 0$ for all $v \in \mathbf{V}(E)$ and therefore (4.9) holds. If v_1 is a unary operator node with outgoing edge e_l, then by applying the induction hypothesis we get

$$w(e) = \frac{c_f(v_1,e)}{c_f(v_1)} w(v_1) = \frac{c_f(v_1,e)}{c_f(v_1)} + \frac{c_f(v_1,e)}{c_f(v_1)} \sum_{v \in \mathbf{V}(E)} \frac{c_f(v,e_l)}{c_f(v)} \tag{4.10}$$

For two nodes u, v let $\mathcal{P}(u,v)$ be the set of paths from u to v in E. Each root path containing e_l must contain v_1 as well, therefore these paths can be split into a root path of v_1 and a path starting from v_1 with edge e_l (cf. Figure 4.6). Furthermore, since v_1 is unary, each root path to a node $v \neq v_1$ that contains e must contain e_l, so these paths can be constructed by combining the root paths to v_1 containing e with the outgoing paths from v_1. It follows that

$$c_f(v_1,e)c_f(v,e_l) = \left(\sum_{\substack{P_0 \in \mathcal{P}(v_1) \,|\\ e \in \mathbf{E}(P_0)}} 2^{\mathrm{cost_f}(P_0)} \right) \left(\sum_{\substack{P \in \mathcal{P}(v) \,|\\ e_l \in \mathbf{E}(P)}} 2^{\mathrm{cost_f}(P)} \right)$$

$$= \sum_{\substack{P_0\in\mathcal{P}(v_1)\,|\\ e\in\mathbf{E}(P_0)}} \sum_{\substack{P\in\mathcal{P}(v)\,|\\ e_l\in\mathbf{E}(P)}} 2^{\mathrm{cost_f}(P_0)+\mathrm{cost_f}(P)}$$

$$= \sum_{\substack{P_0\in\mathcal{P}(v_1)\,|\\ e\in\mathbf{E}(P_0)}} \sum_{P_1\in\mathcal{P}(v_1)} \sum_{P_2\in\mathcal{P}(v_1,v)} 2^{\mathrm{cost_f}(P_0)+\mathrm{cost_f}(P_1)+\mathrm{cost_f}(P_2)}$$

$$= \left(\sum_{\substack{P_0\in\mathcal{P}(v_1)\,|\\ e\in\mathbf{E}(P_0)}} \sum_{P_2\in\mathcal{P}(v_1,v)} 2^{\mathrm{cost_f}(P_0)+\mathrm{cost_f}(P_2)}\right)\left(\sum_{P_1\in\mathcal{P}(v_1)} 2^{\mathrm{cost_f}(P_1)}\right)$$

$$= c_f(v,e)c_f(v_1)$$

Since $c_f(v_1, e_l) = 0$, substituting this equation into (4.10) leads directly to

$$w(e) = \frac{c_f(v_1,e)}{c_f(v_1)} + \sum_{v\in\mathbf{V}(E),v\neq v_1} \frac{c_f(v,e)}{c_f(v)} = \sum_{v\in\mathbf{V}(E)} \frac{c_f(v,e)}{c_f(v)}$$

If v_1 is a binary operator node with outgoing edges e_l, e_r, the proof is similar to the unary case. Analogously, we show that

$$c_f(v_1,e)(c_f(v,e_l) + c_f(v,e_r)) = c_f(v,e)c_f(v_1)$$

holds for all nodes $v \neq v_1$. Since now a path to v that contains e must contain either e_l or e_r, the sum $c_f(v,e)$ consists of both the paths through e_l and the paths through e_r.

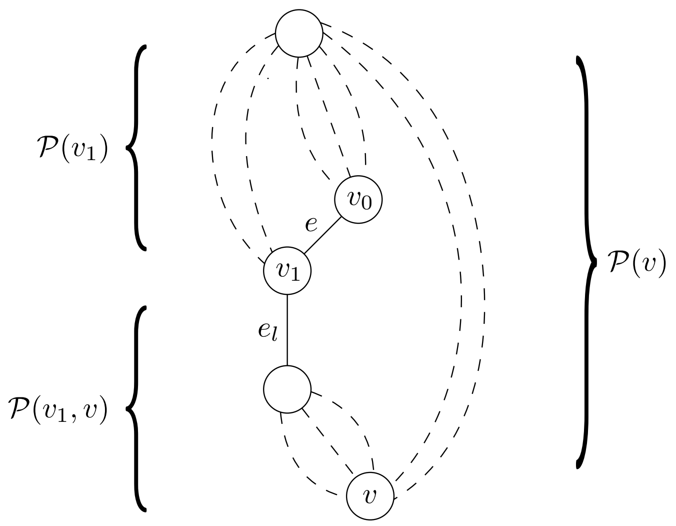


Figure 4.6: The sets $\mathcal{P}(v_1)$, $\mathcal{P}(v_1,v)$ and $\mathcal{P}(v)$ with respect to the edges e, e_l as they appear in the unary case in the proof of Lemma 4.29.

We refrain from explicitly stating the details of this continued equality. Similar to (4.10) the definition of the path weight for binary operator nodes leads to

$$w(e) = \frac{c_f(v_1, e)}{c_f(v_1)} + \frac{c_f(v_1, e)}{c_f(v_1)} \sum_{v \in \mathbf{V}(E)} \frac{c_f(v, e_l) + c_f(v, e_r)}{c_f(v)} = \sum_{v \in \mathbf{V}(E)} \frac{c_f(v, e)}{c_f(v)}$$

completing the induction. □

Based on the path weight, we define an error distribution in which the required accuracies are distributed according to the weight of the respective nodes and edges. Note that the effective weight distributed to each operation is one.

Definition 4.30. Let E be an expression dag that should be evaluated with target accuracy z. Let q be the error distribution in E where $q(E) = z$ and for each node $v \in \mathbf{V}(E)$ and each edge $e = (u, v) \in \mathbf{E}(E)$ we set

$$\begin{aligned} q(v) &= r(v) - \log(w(v)) \\ q(e) &= r(u) + \log(w(e)) - \log(w(u)) - \log(c(e)) \end{aligned}$$

path weight error distribution

We call q the *path weight error distribution* (for E and z).

The resulting error distribution fulfills the necessary requirement for an optimal error distribution that is stated in Lemma 4.26.

Lemma 4.31. Let E be an expression dag with the path weight error distribution. Then each path in E leading to a node $v \in \mathbf{V}(E)$ is defining.

Proof. Let $P = (v_0, e_0, ..., v_k, e_k, v_{k+1})$ be a path in E. Since

$$\begin{aligned} c_f(v_{i+1}, e_i) &= \sum_{P_i \in \mathcal{P}(v_i)} 2^{\mathrm{cost_f}(P_i) + \log(c(e_i))} \\ &= c(e_i) \sum_{P_i \in \mathcal{P}(v_i)} 2^{\mathrm{cost_f}(P_i)} \\ &= c(e_i) c_f(v_i) \end{aligned} \tag{4.11}$$

the accuracy increase due to P is given as

$$\begin{aligned} i(P) &= \sum_{i=0}^{k} i(e_i) + i(v_{k+1}) \\ &= \sum_{i=0}^{k} \Big(\log(w(e_i)) - \log(w(v_i)) - \log(c(e_i)) \Big) - \log(w(v_{k+1})) \\ &= \sum_{i=0}^{k} \Big(\log(w(e_i)) - \log(w(v_{i+1})) - \log(c(e_i)) \Big) - \log(w(v_0)) \\ &= \sum_{i=0}^{k} \Big(\log(c_f(v_{i+1}, e_i)) - \log(c_f(v_{i+1})) - \log(c(e_i)) \Big) - \log(w(v_0)) \end{aligned}$$

$$= \sum_{i=0}^{k} \Big(\log(c_f(v_i)) - \log(c_f(v_{i+1})) \Big) - \log(w(v_0))$$
$$= \log(c_f(v_0)) - \log(c_f(v_{k+1})) - \log(w(v_0)) \tag{4.12}$$

Since $i(P)$ only depends on v_0 and v_{k+1}, the accuracy increase along each root path to v_{k+1} is the same. For each node there is at least one defining root path (cf. Corollary 4.13), therefore each root path in E is defining. Each edge $e = (u, v) \in \mathbf{E}(E)$ lies on at least one root path to v. With Lemma 4.12 it follows that $q(e) = r(v)$ and therefore each path in E is defining. □

Before we show that the path weight error distribution is indeed optimal, we show that the distribution adequately reflects the weight function. In particular, a change in the required accuracy at a node requires a change in the accuracies of the outgoing edges according to their weights if the balancing constraint should be kept equal (cf. Figure 4.7).

Lemma 4.32. Let E be an expression dag and let $e = (u, v)$ be an edge in $\mathbf{E}(E)$. For an error distribution fulfilling the balancing constraint (4.8) at u with equality, let $f_u : (-\infty, 0) \to (-\infty, 0)$ describe the accuracy increase at u as a function of the accuracy increase at e while keeping the balancing constraint equal. Then for the path weight error distribution, $\frac{d}{dx} f_u(x)\,|_{i(e)} = -w(e)$.

Proof. Let $c_e = c(e)$ denote the operator constant at e. If u is unary, then the balancing constraint leads to

$$\frac{d}{dx} f_u(x) = \frac{d}{dx} \log(1 - c_e 2^x) = \frac{-c_e 2^x}{1 - c_e 2^x}$$

By the definition of the path weight error distribution we have

$$c_e 2^{i(e)} = c_e 2^{\log(w(e)) - \log(w(u)) - \log(c_e)} = \frac{w(e)}{w(u)} \tag{4.13}$$

Substituting (4.13) into the derivative and applying the definition $w(u) = 1 + w(e)$ of the path weight for unary operator nodes results in

$$\frac{d}{dx} f_u(i(e)) = \frac{-c_e 2^{i(e)}}{1 - c_e 2^{i(e)}} = \frac{-\frac{w(e)}{w(u)}}{1 - \frac{w(e)}{w(u)}} = \frac{-w(e)}{w(u) - w(e)} = -w(e)$$

If u is binary with outgoing edges $e, \hat{e} \in \mathbf{E}(E)$, then the increase at u depends on both the increase at e and the increase at $\hat{e}$. Let $c_{\hat{e}} = c(\hat{e})$ and let $\hat{f}_u$ be the respective function for the increase at u. We get

$$\frac{\partial}{\partial x} \hat{f}_u(x, y) = \frac{\partial}{\partial x} \log(1 - c_e 2^x - c_{\hat{e}} 2^y) = \frac{-c_e 2^x}{1 - c_e 2^x - c_{\hat{e}} 2^y}$$

For f_u we assume that $y = i(\hat{e})$ is kept constant and therefore $f_u(x) = \hat{f}_u(x, i(\hat{e}))$. As in the unary case, by substituting (4.13) and considering $w(u) = 1 + w(e) + w(\hat{e})$, it follows that

$$\frac{d}{dx} f_u(i(e)) = \frac{-c_e 2^{i(e)}}{1 - c_e 2^{i(e)} - c_{\hat{e}} 2^{i(\hat{e})}} = \frac{-\frac{w(e)}{w(u)}}{1 - \frac{w(e)}{w(u)} - \frac{w(\hat{e})}{w(u)}} = \frac{-w(e)}{w(u) - w(e) - w(\hat{e})} = -w(e) \qquad \square$$

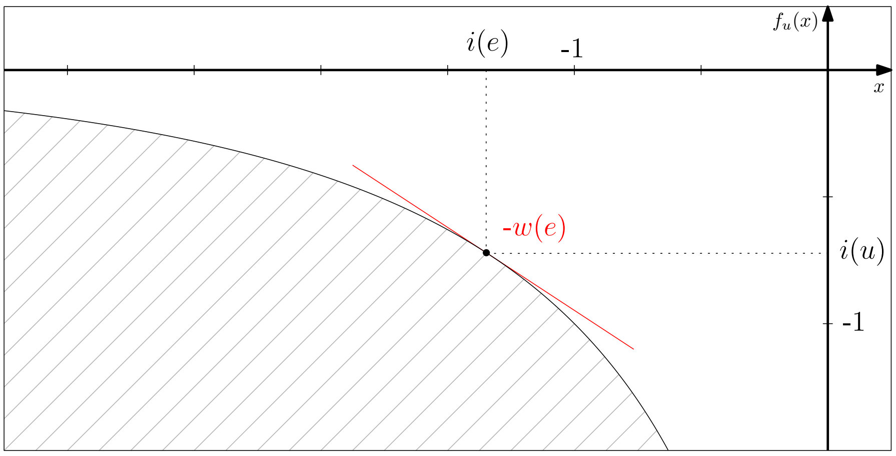


Figure 4.7: The relation of $i(u)$ and $i(e)$ for a unary node u with outgoing edge $e \in \mathbf{E}(E)$ and $c(e) = 1$. The values below the function fulfill the balancing constraint. The path weight error distribution ensures that the slope at the chosen value is equal to the negative path weight of e. Since the slope is monotonically decreasing, an increase by ε in $i(e)$ equates to a decrease of more than $w(e) \cdot \varepsilon$ in $i(u)$ and vice versa.

The path weight error distribution fulfills the balancing constraint with equality. In particular, $i(u) = f_u(i(e))$. If $i(u)$ increases, the required accuracies at its outgoing edges must compensate for the change in order to fulfill the balancing constraint. Since the slope of f_u is strictly decreasing, the change at each outgoing edge e is at least $w(e)$ times as expensive as the change in $i(u)$ that it compensates. If, on the other hand, $i(u)$ decreases, then the potential increase of the required accuracies at the edges is bounded by the balancing constraint and thereby bounded by $w(e)$ times the change in $i(u)$ for an outgoing edge e. With this observation, we can show that the path weight error distribution is optimal. For the proof, we show that the path weight accurately reflects the impact that a change in accuracy at a given edge has on the total cost of the evaluation by showing that the total cost difference for any change of the error distribution is positive.

Theorem 4.33. Let E be an expression dag. The path weight error distribution is the uniquely optimal error distribution in E for each target accuracy z.

Proof. Let q denote the path weight error distribution for E and z and let q^* be another error distribution for E with $q^*(E) = q(E)$. For $v \in \mathbf{V}(E)$ and $e \in \mathbf{E}(E)$

let $\delta(v) = i^*(v) - i(v)$ and $\delta(e) = i^*(e) - i(e)$ be the differences in the accuracy increases between q^* and q. Due to Lemma 4.32, the difference in the required accuracy for the operation can be bounded by the differences in the required accuracies of its outgoing edges. We have $\delta(v) \leq 0$ if v is a quasi-leaf and

$$\delta(v) \leq \begin{cases} -w(e_l)\delta(e) & \text{if } v \text{ is quasi-unary with outgoing edge } e_l \in \mathbf{E}(E) \\ -w(e_l)\delta(e_l) - w(e_r)\delta(e_r) & \text{if } v \text{ is fully binary with outgoing edges } e_l, e_r \end{cases}$$

In particular, since each edge has exactly one origin, we get

$$\sum_{v \in \mathbf{V}(E)} \delta(v) \leq - \sum_{e \in \mathbf{E}(E)} w(e)\delta(e) \tag{4.14}$$

The total cost difference between the two error distributions can be characterized by the root paths as in (4.5). For a path P in E, denote the difference in cost between q^* and q by $\delta(P) = i^*(P) - i(P)$. Since each path in E is defining for a path weight error distribution, the cost difference between q^* and q is given as

$$\begin{aligned} \Delta \operatorname{cost}(E) &= - \sum_{v \in \mathbf{V}(E)} \left(\min_{P \in \mathcal{P}(v)} i^*(P) - \min_{P \in \mathcal{P}(v)} i(P) \right) \\ &= - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \delta(P) \end{aligned}$$

By applying the definition of $i(P)$ together with (4.14) and Lemma 4.29, we get

$$\begin{aligned} \Delta \operatorname{cost}(E) &= - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \delta(P) \\ &= - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \left(\sum_{e \in \mathbf{E}(P)} \delta(e) + \delta(v) \right) \\ &= - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \sum_{e \in \mathbf{E}(P)} \delta(e) - \sum_{v \in \mathbf{V}(E)} \delta(v) \\ &\geq - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \sum_{e \in \mathbf{E}(P)} \delta(e) + \sum_{e \in \mathbf{E}(E)} w(e)\delta(e) \\ &= - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} \sum_{e \in \mathbf{E}(P)} \delta(e) + \sum_{e \in \mathbf{E}(E)} \delta(e) \sum_{v \in \mathbf{V}(E)} \frac{c_f(v,e)}{c_f(v)} \end{aligned}$$

The sum of $c_f(v, e)$ over all edges is the same as the sum of $2^{\operatorname{cost_f}(P)}$ over the edges of all paths leading to v (cf. Figure 4.8). By interchanging sums, by bounding the cost difference caused by accuracy increases along the edges of a path from below and with the definitions of $c_f(v, e)$ and $c_f(v)$, we finally get

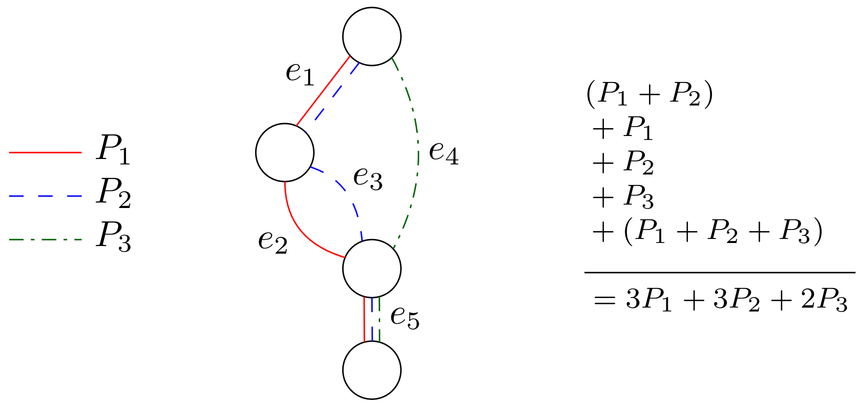


Figure 4.8: Schematic depiction of the equivalence between the sum over all paths through all edges and the sum over all edges in all paths.

$$
\begin{aligned}
\Delta\operatorname{cost}(E) &\geq -\sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) + \sum_{e\in\mathbf{E}(E)} \delta(e) \sum_{v\in\mathbf{V}(E)} \frac{c_f(v,e)}{c_f(v)} \\
&= -\sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) + \sum_{v\in\mathbf{V}(E)} \sum_{e\in\mathbf{E}(E)} \delta(e) \sum_{P\in\mathcal{P}(v)\,|\,e\in\mathbf{E}(P)} \frac{2^{\operatorname{cost}_f(P)}}{c_f(v)} \\
&= -\sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) + \sum_{v\in\mathbf{V}(E)} \sum_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) \frac{2^{\operatorname{cost}_f(P)}}{c_f(v)} \\
&\geq -\sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) + \sum_{v\in\mathbf{V}(E)} \left(\min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) \right) \sum_{P\in\mathcal{P}(v)} \frac{2^{\operatorname{cost}_f(P)}}{c_f(v)} \\
&= -\sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) + \sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} \sum_{e\in\mathbf{E}(P)} \delta(e) \;=\; 0
\end{aligned}
$$

and therefore q is optimal. Since (4.14) is equal only if $\delta(e) = 0$ for all e, the path weight error distribution is the unique optimal error distribution for E. □

Corollary 4.34. The cost of evaluating an expression dag E with n operator nodes and an optimal error distribution to accuracy $z \leq 0$ is

$$\operatorname{cost}(E) = n\log n + \sum_{v\in\mathbf{V}(E)} \log\left(\sum_{P\in\mathcal{P}(v)} 2^{\operatorname{cost}_f(P)} \right) - nz$$

Proof. If v_0 is the root of E, then $c_f(v_0) = 1$ and $w(v_0) = n$. Since the path weight error distribution is optimal, the desired result follows directly from (4.5) and (4.12) as

$$\operatorname{cost}(E) = -nz - \sum_{v\in\mathbf{V}(E)} \min_{P\in\mathcal{P}(v)} i(P) = -nz - \sum_{v\in\mathbf{V}(E)} (-\log(c_f(v)) - \log n)$$

□

4.2.2 Implementation

In an implementation of an exact-decisions number type using the standard error distribution (Definition 4.18, page 55), the required accuracies at each node can be computed on the fly during the evaluation. If a path weight error distribution (Definition 4.30, page 62) is used, a preprocessing step is required. In practice, the required and requested accuracies are computed during the evaluation and only the requested accuracies, in case of a topological evaluation strategy, or no accuracies at all, in case of a recursive evaluation strategy, are stored at a node (cf. Section 3.2.2). In order to focus on the error

Algorithm 1: Algorithm for computing the requested accuracies for each node during the evaluation of an expression dag E with target accuracy z, given that the increase values for each node and each edge are known and the requested accuracies are initialized with ∞.

```
Function compute_requested_accuracies($E$,$z$):
    $V[1..n]$ = top_sort($\mathbf{V}(E)$);
    $r(v_1) = z$;
    for $j = 1$ to $n$ do
        $q(v_j) = r(v_j) + i(v_j)$;
        forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
            $q(e) = r(v_j) + i(e)$;
            $r(v_k) = \min(r(v_k), q(e))$;
        end
    end
```

Algorithm 2: Algorithm for computing the standard error distribution for an expression dag E in the form of increase values for each node and each edge.

```
Function compute_standard_error_distribution($E$):
    forall $v \in \mathbf{V}(E)$ do
        $i(v) = -1$;
        if $v$ is a quasi-leaf then
            $i(v) = 0$
        else if $v$ is quasi-unary then
            $e \leftarrow$ outgoing edge of $v$ in $\mathbf{E}(E)$;
            $i(e) = -1 - \lceil \log(c(e)) \rceil$;
        else
            $e_l, e_r \leftarrow$ outgoing edges of $v$;
            $i(e_l) = -2 - \lceil \log(c_l) \rceil$;
            $i(e_r) = -2 - \lceil \log(c_r) \rceil$;
        end
    end
```

distribution, we describe the necessary steps for computing the increase values at each node in a separate algorithm. The requested and required accuracies at each node can then be computed using a topological sorting as depicted in Algorithm 1. The increase values for the standard error distribution can be computed in a straightforward manner. Algorithm 2 shows an exemplary implementation. Each increase value in this distribution has an integral value and therefore both the increase values and the resulting error bounds can be stored in an integer data type. For the path weight error distribution, a topological sorting of the operator nodes is required. As shown in Algorithm 3, we compute the values for $c_f(v_k, e)$ and $c_f(v_k)$, the weights $w(v_k)$ and the increase values at v_k separately for each node v_k and edge $e = (v_j, v_k)$. The values of $c_f(v_k, e)$ and $c_f(v_k)$ are computed by traversing the operator nodes top-down. Equation (4.11) on page 62 provides us with an inductive formula to compute $c_f(v_k, e)$. Furthermore, from its definition we can easily

Algorithm 3: Algorithm for computing the path weight error distribution for an expression dag E in the form of increase values for each node and each edge. The algorithm first computes the values $c_f(v)$ and $c_f(v, e)$ for each node and each incoming edge top-down. Afterwards it computes the path weight and the increase values for each node and each edge bottom-up.

```
Function compute_path_weight_error_distribution($E$):
    $V[1..n]$ = top_sort($\mathbf{V}(E)$);
    forall $v \in V$ do
        $c_f(v) = 0$; $w(v) = 1$;
    end
    $c_f(v_1) = 1$;
    for $j = 1$ to $n$ do
        forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
            $c_f(v_k, e) = c_f(v_j)c(e)$;
            $c_f(v_k) = c_f(v_k) + c_f(v_k, e)$
        end
    end
    for $j = n$ downto $1$ do
        forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
            $w(e) = \frac{c_f(v_k,e)}{c_f(v_k)} w(v_k)$;
            $w(v_j) = w(v_j) + w(e)$;
        end
        $i(v_j) = -\log(w(v_j))$;
        forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
            $i(e) = \log\left(\frac{w(e)}{c(e)w(v_j)}\right)$;
        end
    end
```

deduce that $c_f(v_k)$ is given as the sum of $c_f(v_k, e)$ over all of its incoming edges. After computing $c_f(v)$ and $c_f(v, e)$ for each node and each of its incoming edges, both the weights of the nodes and edges and their increase values are computed according to their respective definition by traversing the operator nodes bottom-up. All values appearing in the algorithm for the path weight error distribution are rational numbers that are usually non-integral. Unlike before, floating-point number types must be used to represent these values, introducing rounding errors. The resulting error distribution is guaranteed to be valid as long as the weight of a node is at least as large as the sum of the weights of its outgoing edges plus one. This can be guaranteed if we ensure that the addition in line 16 is always rounded up and the computations of the increase values in line 18 and line 20 are (in total) always rounded down. The values of $c_f(v_k, e)$ and $c_f(v_k)$ can get very large if many common subexpressions and therefore many paths exist. Therefore, they should be evaluated and stored logarithmically. In consequence, the weight values should be stored logarithmically as well. While this decreases the susceptibility to errors for multiplications and divisions, precautionary measures need to be taken to compute the additions in line 10 and line 16. We address this issue in Section 6.1.

4.2.3 Weight Heuristics

The path weights of the edges and nodes in an expression dag E depend on the operator constants of the edges. Since the values of those constants can differ greatly in magnitude, rounding errors can have a significant influence on their computation. This raises the question whether the weight function can be replaced by a heuristic that can be computed faster and is less prone to rounding errors. We first observe that we can apply the construction from the definition of the path weight error distribution (cf. Definition 4.30) to almost every weight function to get a valid error distribution.

Observation 4.35. Let E be an expression dag and let w be a function mapping each edge $e \in \mathbf{E}(E)$ to a weight $w(e) > 0$ and each node $v \in \mathbf{V}(E)$ to a weight $w(v) \geq 1 + \sum_{(v,v') \in \mathbf{E}(E)} w((v, v'))$. Then for any target accuracy z the error distribution q defined by $q(E) = z$ and

$$\begin{aligned} q(v) &= r(v) - \log(w(v)) \\ q(e) &= r(v) + \log(w(e)) - \log(w(v)) - \log(c(e)) \end{aligned} \tag{4.15}$$

for $v \in \mathbf{V}(E)$ and $e = (u, v) \in \mathbf{E}(E)$ is valid.

The computation of the path weight error distribution simplifies if the expression dag E has few common subexpressions. If E is a tree, the path weight of a node is equivalent to the number of operator nodes in its subexpression.

Definition 4.36. For an expression dag E, we call $|\mathbf{V}(E)|$ the *operator number* of E and denote it by $|E|_\circ$. $|E|_\circ$

Lemma 4.37. Let T be an expression tree. Let $e = (u, v) \in \mathbf{E}_0(T)$ with $u, v \in \mathbf{V}_0(T)$ and let T_v be the subexpression rooted at v. Then the path weight of e and v is given as $w(e) = w(v) = |T_v|_\circ$ and the cost of the evaluation of T with the path weight error distribution to accuracy z is

$$\mathrm{cost}(T) = n \log n + \sum_{v \in \mathbf{V}(T)} \mathrm{cost}_\mathrm{f}(\mathrm{path}(v)) - nz$$

where $\mathrm{path}(v)$ denotes the unique path in $\mathcal{P}(v)$.

Proof. Since T does not contain common subexpressions, there is only one root path, called $\mathrm{path}(v)$, leading to a node $v \in \mathbf{V}_0(T)$. By the definition of the path weight, we get

$$c_f(v, e) = \begin{cases} c_f(v) & \text{if } e \in \mathbf{E}(\mathrm{path}(v)) \\ 0 & \text{otherwise} \end{cases}$$

for all edges $e \in \mathbf{E}_0(T)$ and $w(e_v) = w(v)$ for $e_v = (u, v)$. With Lemma 4.29 (page 60) it follows that $w(v) = w(e_v) = |\mathbf{V}(T_v)|$. The cost for the evaluation follows directly from the general cost formula in Corollary 4.34 with

$$\log\left(\sum_{P \in \mathcal{P}(v)} 2^{\mathrm{cost}_\mathrm{f}(P)}\right) = \log\left(2^{\mathrm{cost}_\mathrm{f}(\mathrm{path}(v))}\right) = \mathrm{cost}_\mathrm{f}(\mathrm{path}(v))$$

□

Building on these observations, we define a weight function with regard to the number of operator nodes in a node's subexpression.

Definition 4.38. Let E be an expression dag and let E_v be the subexpression of E rooted at $v \in \mathbf{V}_0(E)$. Let $w(v) = |E_v|_\circ$ and for an edge $e = (u, v) \in \mathbf{E}_0(E)$ let

$$w(e) = \begin{cases} w(v), & \text{if } u \text{ is unary} \\ \frac{w(v)+w(u)-1-w(v')}{2}, & \text{if } u \text{ is binary with outgoing edges } e \text{ and } e' = (u, v') \end{cases}$$

operator weight

Then w is called the *operator weight* of v and e.

In a binary node, the operator weight of an edge counts the number of operator nodes in the subexpression of its associated child, except for nodes that appear in both child expressions. These nodes contribute half of their weight to each edge. Lemma 4.37 suggests that for tree-like expression dags, using the operator weight instead of the path weight to determine an error distribution leads to a reasonable heuristic with respect to the cost of the evaluation. All weight values of w are integral or end with .5 and are smaller than the size of the expression dag. Therefore they can be represented exactly. Unfortunately, computing the exact number of descendants for each node in a dag is hard. The weights can be found in slightly less than quadratic time by computing the transitive closure of the dag [BVW08]. Nevertheless, a computation in time $O(m^{2-\epsilon})$, where m is the number of edges in the dag, is not possible if the strong exponential time hypothesis is true [Bor16]. Using this weight function in an evaluation is therefore not feasible if the operator number is not already known.

If no common subexpressions are present, the operator weight of a node can be computed by simply adding one to the sum of the operator weights of its children. Instead of computing the actual operator weights of a dag, we can virtually expand the dag into a tree and use the operator weights of this tree expansion.

Definition 4.39. Let E be an expression dag. For a node $v \in \mathbf{V}(E)$ and an edge $e = (u, v) \in \mathbf{E}(E)$ let $w(e) = w(v) = 1 + \sum_{(v,v') \in \mathbf{E}(E)} w((v, v'))$. Then we call w the *tree weight* of v and e.

tree weight

The tree weight is similar to the weight function used in the work of van der Hoeven. For a node v, it is equivalent to the number of paths from v to all operator nodes in its subexpression. In contrast to the operator weight, it can be computed with an easy inductive formula. Furthermore, all values of the tree weight are integral. However, similar to the parameters used during the path weight computation, the tree weight can grow exponentially in the size of the expression dag. In Algorithm 4, a possible implementation is sketched. In line 6, a check can be inserted on whether the result still fits into the underlying integer type. Otherwise, as for the path weight error distribution, we can switch to an inexact logarithmic representation. With this additional step, the tree weight can be represented exactly for expression dags with few common subexpressions.

Algorithm 4: Algorithm for computing an error distribution for E based on the tree weight in the form of increase values for each node and each edge. The weights and increase values are computed in a single bottom-up loop.

```
1  Function compute_tree_weight_error_distribution($E$):
2      $V[1..n]$ = top_sort($\mathbf{V}(E)$);
3      for $j = n$ downto $1$ do
4          $w(v_j) = 1$;
5          forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
6              $w(v_j) = w(v_j) + w(v_k)$;
7          end
8          $i(v_j) = -\log(w(v_j))$;
9          forall $e = (v_j, v_k) \in \mathbf{E}(E)$ do
10             $i(e) = \log\left(\frac{w(v_k)}{c(e)w(v_j)}\right)$;
11         end
12     end
```

For an expression tree, the path weight, the operator weight and the tree weight are identical. Otherwise, an error distribution based on the operator weight usually leads to a lower evaluation cost than an error distribution based on the tree weight. In Figure 4.9, the behavior of the three error distributions is shown for a simple example. The tree weight leads to the same required accuracy at each operator node, overestimating the impact of an accuracy increase along the edges on the total cost of the evaluation. The operator weight still overestimates the impact of the edges but can partly compensate for multiple references inside a subexpression. In consequence, it computes the correct

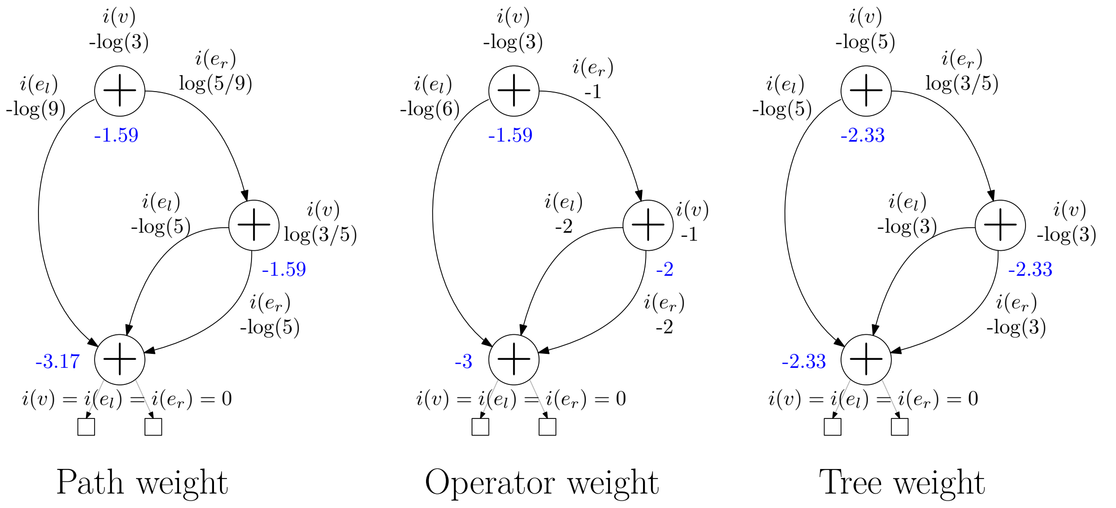


Figure 4.9: A comparison of the three weight functions for error bound balancing. For each node, the increase values at the node and at the two outgoing edges are stated. The total required accuracy at a node is shown in blue (for $q(E) = 0$). The path weight leads to the minimum total cost of 6.35. The operator weight and the tree weight lead to a total cost of 6.59 and 6.99, respectively.

increase value for the root node. However, it fails to recognize dependencies outside the subexpression as well as differences in the strength of a dependency inside a subexpression. Nevertheless, all strategies reduce the cost compared to the total cost of 8 induced by the standard error distribution. If the number of common subexpressions is low, it may therefore be worthwhile to employ one of the heuristic approaches instead of the more complicated path weight.

4.3 Restructuring

In the previous section, it was shown how error bounds in an accuracy-driven evaluation process can be distributed to mitigate the impact of badly balanced expression dags. With error bound balancing, the variable increase in error can be made independent of the structure. However, the operator constants associated with the accuracy increase still lead to an increased evaluation cost for unbalanced expression dags.

In this section we aim to restructure the expression dag to facilitate an efficient evaluation. In the first subsection, the general framework for restructuring is described. We classify restructuring algorithms by their *invasiveness*. In Section 4.3.2, a simple non-invasive restructuring algorithm is introduced and it is shown that this algorithm can effectively reduce the cost of large sums and large products. Furthermore, various concepts for semi-invasive restructuring methods are discussed. In Section 4.3.3, a powerful restructuring strategy, based on an algorithm by Richard Brent, is introduced. This strategy is classified as invasive. In the main theorem of this section, we show that Brent Restructuring has a degree of invasiveness of 4. In Section 4.3.4, the previously introduced restructuring methods are made responsive to weights associated with the nodes in an expression dag. Afterwards, we show that these weighted methods can improve on the previous restructuring algorithms if nodes that inhibit the restructuring process, so-called *blocking nodes*, are present in the expression dag.

4.3.1 Preliminaries

Restructuring expression dags for exact computation was proposed by Chee Yap in 1997 as a side note on possible improvements for number types implementing accuracy-driven evaluation [Yap97]. While present for a long time, there have been no publications regarding this topic prior to this work. One reason for that might be that the effect for small computations seems not worth the effort. Furthermore, aside from the fundamental requirement that an evaluation must store the computation history in the form of an expression dag, restructuring demands fairly strong properties for the expression dag. In this section, we define the scope for the restructuring algorithms presented in this work.

Pre-Evaluated Nodes

In general, meaningful implementations of restructuring strategies depend on the lazy evaluation approach, which is inherent to accuracy-driven evaluation. If the expression dag has not been evaluated before, internal data such as bigfloat approximations have not been computed. Consequently, changing the structure of such an expression dag comes at low cost. If (a part of) the expression dag has already been evaluated, approximation data and error bounds have been computed that are usually worthless after restructuring. Furthermore, internal states that might have been set during the initialization of the bigfloat data are invalidated. Restructuring already evaluated parts of an expression dag might be worthwhile if we can guarantee that most of the existing bigfloat data must be recomputed anyway and that the evaluation significantly benefits from a new structure. For this work, however, we focus on an analysis of the fundamental properties of restructuring methods and therefore refrain from restructuring parts of an expression dag that have been evaluated before.

Common Subexpressions

An expression dag may contain common subexpressions, i.e., subexpressions whose root node has more than two parents. Including those nodes in a restructuring process implies that the associated subexpressions must be copied. There are few cases in which this can be considered a sensible choice. The benefit of the improved restructuring has to be larger than the cost of an additional evaluation. This can be the case if one parent needs to evaluate the subexpression often and with much higher accuracy than the other parent or if a node prevents large parts of the expression to be represented exactly. The potential effect of relocating such nodes is demonstrated in Section 4.4.3. Careless copying of subexpressions, on the other hand, may lead to an exponential increase of the number of

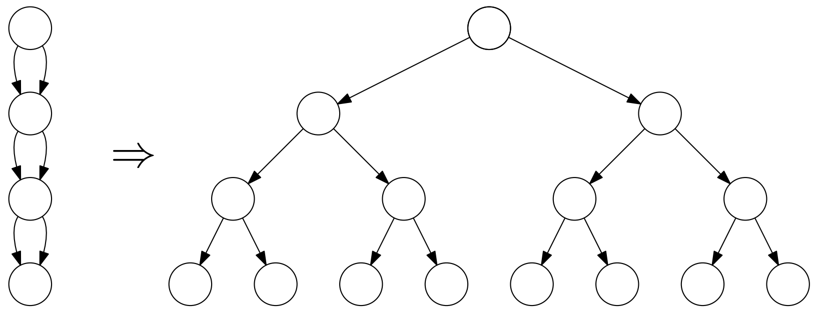


Figure 4.10: The tree expansion of an expression dag with a high number of common subexpressions. Resolving multiple references by copying the respective subexpressions leads to an exponential increase in the number of operator nodes.

nodes as shown in Figure 4.10. In this work we do not follow this path and therefore do not use nodes with more than one parent for restructuring.

Remark 4.40. In contrast to Section 4.2, we cannot unconditionally assume that the expression dag in question is rooted since the existence of multiple parents is not restricted to the local evaluation (cf. Remark 2.9, page 22). For now, we avoid lengthy reformulations of various definitions by assuming that these nodes are marked as not eligible for restructuring and that an actual implementation will handle them correctly. Nevertheless, in Section 4.4 we exploit these external references as a simple way to create “blocking” nodes for an experiment. We elaborate on the handling of external references and, more generally, blocking nodes in Section 4.3.4.

Neglecting nodes with multiple parents for restructuring implies that we are limited to restructure expression trees. With the introduction of the concept of *operator trees*, we can focus our restructuring efforts on independent parts of a larger expression dag.

Operator Trees

We start by identifying parts of an expression dag that consist exclusively of operations.

operator dag **Definition 4.41.** Let E be an expression dag and let X be a rooted connected subgraph of E where every node is an operator node. Then we call X an *operator dag* (in E).

Let X be an operator dag with k outgoing edges in E. By imposing an order on the edges, X can be interpreted as a k-ary operation, which is independent of E. We denote the expression dag created by applying X to expression dags $E_1, ..., E_k$ as $X(E_1, ..., E_k)$.
operands, $|X|_l$ We call $E_1, ..., E_k$ the *operands* of X and denote their number by $|X|_l$. If the operands of an operator dag are not specified explicitly, we associate an ordered set of virtual operands with it. The operation defined by the operator dag is then applied by identifying the actual operands with the virtual ones. With this we can define the concept of a restructuring algorithm.

Definition 4.42. Let $\mathcal{D}$ be a set of operator dags. Let R be a function that maps each operator dag $X \in \mathcal{D}$ with $|X|_l = k$ to an operator dag $R(X)$ with the same number of operands. Then we call R a *restructuring algorithm* if for all expression dags $E_1, ..., E_k$ it follows that restructuring alg.

$$\text{value}(R(X)(E_1, ..., E_k)) = \text{value}(X(E_1, ..., E_k))$$

The set $\mathcal{D}$ is called the *domain* of R. domain

At the start of this section, we specified what parts of an expression dag are eligible for restructuring. For simplicity, we assume that all expression dags appearing in this section do not contain pre-evaluated nodes. Furthermore, we only consider restructuring algorithms that work on trees.

Definition 4.43. Let E be an expression dag and let T be an operator dag in E. We call T an *operator tree* in E if every node in T, except for its root, has exactly one parent in E (and no external references). operator tree

With this property, an operator tree cannot contain any cycles. Since each operator dag is connected, it follows:

Observation 4.44. Each operator tree in an expression dag E is a tree.

All of the restructuring strategies that we describe in this section operate on (maximal) operator trees inside an expression dag. Obviously, this implies that there are unbalanced expression dags that cannot be restructured by these methods. The extraction of operator trees from an expression dag as well as the interaction between the local restructuring methods and the overall expression dag are addressed in Section 4.3.4. For now, we assume that we work on already extracted operator trees.

4.3.2 Non-Invasive Restructuring Methods

Restructuring an expression dag should not lead to an increase in running time. Rearranging a formula may increase the number of operations needed to arrive at its value. If the number of operations is increased considerably, the evaluation of the restructured expression may perform significantly worse, even if it is more balanced than the original. This leads to the definition of *invasiveness.*

Definition 4.45. Let R be a restructuring algorithm with domain $\mathcal{D}$. We call R *non-invasive* if for each $X \in \mathcal{D}$ the resulting operator dag $R(X)$ contains the same operators as X (including their respective number). We call R *semi-invasive* if $|X|_\circ \leq |R(X)|_\circ$, i.e., if restructuring does not lead to an increase in the number of operator nodes. Otherwise, we call R *invasive.* non-invasive semi-invasive invasive

An invasive restructuring algorithm increases the total number of bigfloat operations necessary for the evaluation. If the target accuracy is high, the cost of the evaluation increases through restructuring. For semi-invasive restructuring algorithms, the difference in cost is never negatively affected by a high target accuracy. With a non-invasive restructuring algorithm it can furthermore be expected that there is no increase in running time due to the nature of the bigfloat operations.

AM-Balancing

A simple approach on a non-invasive restructuring algorithm is to exploit associativity in additions and multiplications. Large operator trees consisting of only additions or only multiplications can be replaced by balanced versions of these trees [Wil17]. We call this method *AM-Balancing.*

AM-Balancing

Algorithm 5: The main steps in AM-Balancing.

```
    Precondition: $T$ contains only nodes with operator $\circ \in \{+, \cdot\}$
1   Function AM_Balance($T$):
2       $X[1..n]$ = get_operands($T$);
3       for $i = 1$ to $2n - 3$ by 2 do
4           $X[n + \frac{i+1}{2}]$ = make_node($X[i]$,$X[i+1]$,$\circ$);
5       end
6       return $X[2n - 1]$;
```

Algorithm 5 shows an exemplary implementation of AM-Balancing. Operator trees are restructured by discarding all operators and building a new, balanced, operator tree with the same number of operations. Let E be an expression dag consisting of n additions over floating-point numbers. In Section 4.1.3, it was shown that this restructuring strategy always leads to an absolute cost of order $n \log n - nz$ if E is evaluated with the standard error distribution and target accuracy z. More generally, let

$\mathrm{cost}_{\mathrm{v}}(E)$

$$\mathrm{cost}_{\mathrm{v}}(E) = - \sum_{v \in \mathbf{V}(E)} \min_{P \in \mathcal{P}(v)} (i(P) + \mathrm{cost}_{\mathrm{f}}(P))$$

variable cost

be the *variable cost* of an expression dag E with error distribution q, i.e., the cost caused by the accuracy increase along its edges that is not inflicted by the operator constants. Then AM-Balancing reduces the variable cost of eligible expression trees to a minimum. More formally, denote AM-Balancing by AM and let $\mathcal{D}$ be the domain of AM. Let $T \in \mathcal{D}$ with $|T|_l = k$ and let $x_1, ..., x_k \in \mathbb{R}$. Then with the standard error distribution,

$$\mathrm{cost}_{\mathrm{v}}(AM(T)(x_1, ..., x_k)) \leq \mathrm{cost}_{\mathrm{v}}(T(x_1, ..., x_k))$$

Instead of a, rather technical, proof, we give a short intuition. All perfectly balanced expression trees have the same cost. Let T be an unbalanced expression tree and let X be one of the smallest subexpressions for which the maximum depth of the left side is larger than the minimum depth of the right side plus one (otherwise we switch sides). Then by attaching one of the lowest nodes on the left side to the right side we can reduce the variable cost at this node by at least one and the total cost difference is smaller or equal to zero. Repeating this step, we can construct a balanced expression dag with the same or smaller variable cost.

In Section 4.1.2, we introduced two cost functions, the *true cost* and the *absolute cost.* In Section 4.2.1, we argued that we can use the absolute cost for our analysis since the correction addends for converting between the required accuracy at a node and the operator precision are not influenced by the balancing method. Furthermore, we considered the operator-dependent factors in the error distribution to be constant with respect to the balancing method. These assumptions are not applicable anymore if the structure of an expression dag changes. For same sized operands, the maximum size of a subexpression is influenced by the number of nodes in that subexpression and can therefore be affected by restructuring. Consequently, we analyze the effects of AM-Balancing with respect to the true cost. AM-Balancing always generates a perfectly balanced tree. Given a number of same-sized operands, we determine the true cost of a balanced operator tree and the true cost of an operator tree forming a linear list (cf. Figure 4.4, page 56). While not a thorough analysis of all possible operator trees, we can expect the basic findings to transfer naturally to intermediate cases.

Lemma 4.46. Let T_{list} be a list-like expression tree and let T_{bal} be a balanced expression tree, both consisting of n additions and $n+1$ operands. Let m be the maximum magnitude of the operands of T_{list} and T_{bal}. Then the true cost of evaluating these expression trees to accuracy z with the standard error distribution is given as

$$\mathrm{cost_t}(T_{list}) = \Theta(-nz + n^2 + nm) \quad \text{and} \quad \mathrm{cost_t}(T_{bal}) = \Theta(-nz + n\log n + nm)$$

Proof. In the standard error distribution, we have $i(v) = 0$ for each quasi-leaf, and $i(v) = -1$ for each quasi-unary or fully binary node $v \in \mathbf{V}(E)$. For an edge $e \in \mathbf{E}(E)$ we have $i(e) = -1$ if it originates in a quasi-unary addition node and $i(e) = -2$ if it originates in a fully binary addition node.

For $0 \leq j \leq n-1$, let v_j denote the operator node on the j-th level in T_{list}. Each operator node in T_{list} except for v_{n-1} is quasi-unary. The required accuracy at v_j is then given as $q(v_j) = z - j - 1$ for $0 \leq j \leq n-2$ and as $q(v_{n-1}) = z - (n-1)$ otherwise. The maximum magnitude of the value of a sum increases logarithmically with the number of (same-sized) summands, therefore the maximum magnitude at v_j is $m + \lceil \log(n-j+1) \rceil$. The correction addend for an addition is defined as the maximum of the magnitudes of its operands plus one (cf. Table 4.3). Altogether, the precision at v_j is given as

$$\begin{aligned} p(v_j) &\leq -q(v_j) + \max(m + \lceil \log(n-j) \rceil, m) + 1 \\ &\leq -z + j + 1 + m + \lceil \log(n-j) \rceil + 1 \end{aligned}$$

The true cost of an expression dag is defined as the sum of all operator precisions, hence

$$\begin{aligned} \mathrm{cost_t}(T_{list}) &= \sum_{j=0}^{n-1} p(v_j) \\ &\leq \sum_{j=0}^{n-1} (-z + j + 1 + m + \lceil \log(n-j) \rceil + 1) \\ &= -nz + \frac{n(n+1)}{2} + nm + O(n \log n) \end{aligned}$$

and therefore $\mathrm{cost_t}(T_{list}) = O(-nz + n^2 + nm)$. If each summand has maximum value, this bound is strict.

For T_{bal}, each predecessor of an operator node v_j on the j-th level, except for the last level, must be fully binary and therefore

$$z - 2j - 1 \leq q(v_j) \leq z - 2(j-1) - 1$$

Let $k = \lceil \log(n+1) \rceil$ be the number of levels in T_{bal}. In the subtree rooted at v_j, there are between 2^{k-j-1} and 2^{k-j} operands. Therefore, an upper bound to the maximum magnitude at v_j is given as $m + k - j$. Since on each level there are at most 2^j operator nodes, we get

$$\begin{aligned} \mathrm{cost_t}(T_{bal}) &= \sum_{v \in \mathbf{V}(T_{bal})} p(v) \\ &\leq \sum_{j=0}^{k-1} 2^j \left(-z + 2j + 1 + m + k - j\right) \\ &= O\left(-nz + n \log n + nm\right) \end{aligned}$$

We get a matching lower bound by assuming that each summand has maximum value, using the lower bounds in the respective estimates and ignoring the $(k-1)$-th level. □

The above lemma shows that balancing a summation asymptotically reduces the true cost of the expression if the additions previously formed a linear list. With a little more thought, it becomes evident that this bound is optimal among all possible summation strategies. For multiplication, the true cost is generally dominated by the increase in the operator constants.

Lemma 4.47. Let T_{list} be a list-like expression tree and let T_{bal} be a balanced expression tree, both consisting of n multiplications. Let m be the maximum magnitude of the operands of T_{list} and T_{bal}. Then the true cost of evaluating E_{list} or E_{bal} to accuracy z with the standard error distribution is given as

$$\mathrm{cost_t}(T_{list}) = \Theta(-nz + n^2 m) = \mathrm{cost_t}(T_{bal})$$

Proof. As for additions, we have $i(v) = 0$ for each quasi-leaf, and $i(v) = -1$ for each quasi-unary or fully binary node $v \in \mathbf{V}(E)$. For an edge $e \in \mathbf{E}(E)$, we have $i(e) = -1 - \log(c(e))$ if it originates in a quasi-unary node and $i(e) = -2 - \log(c(e))$ if it originates in a fully binary node. Furthermore, the maximum magnitude of a product increases linearly in the number of factors.

As before, let v_j denote the operator node in T_{list} on the j-th level. The maximum magnitude at a node v_j for $0 \leq j \leq n-1$ is then given as $(n-j+1)m$. The operator constant at an edge $e \in \mathbf{E}(E)$ is an upper bound to the size of the second operand, which, in this case, is always a leaf. Therefore, $\log(c(e)) \leq m$. The correction addend of a multiplication is bounded by the size of the resulting approximation, therefore

$$\begin{aligned} p(v_j) &\leq -q(v_j) + (n-j+1)m \\ &\leq -z + j\,(1+m) + 1 + (n-j+1)m \\ &= -z + j + 1 + (n+1)m \end{aligned}$$

Consequently, the true cost is bounded by

$$\begin{aligned} \mathrm{cost_t}(T_{list}) &\leq \sum_{j=0}^{n-1} (-z + j + 1 + (n+1)m) \\ &= -nz + \frac{n(n+1)}{2} + n(n+1)m \end{aligned}$$

For T_{bal}, with $k = \lceil \log(n+1) \rceil$, the maximum magnitude of a node v_j on level j is bounded from above by $2^{k-j}m$ and, if all operands have maximum magnitude, from below by $2^{k-j-1}m$. The operator precision of v_j is therefore bounded by

$$\begin{aligned} p(v_j) &\leq -q(v_j) + 2^{k-j}m \\ &\leq -z + \sum_{i=0}^{j-1} \left(2 + 2^{k-i}m\right) + 1 + 2^{k-j}m \\ &= -z + 2(j-1) + \left(2^{k+1} - 2^{k-j+1} + 2^{k-j}\right) m \\ &= -z + 2(j-1) + \left(2^{k+1} - 2^{k-j}\right) m \end{aligned}$$

For the true cost of T_{bal} it follows that

$$\begin{aligned} \mathrm{cost_t}(T_{bal}) &\leq \sum_{j=0}^{k-1} 2^j \left(-z + 2(j-1) + \left(2^{k+1} - 2^{k-j}\right) m\right) \\ &= O\left(-nz + n^2 m\right) \end{aligned}$$

This yields an asymptotic upper bound for $\mathrm{cost_t}(T_{bal})$. As before, matching lower bounds are shown by assuming that all operands have maximum magnitude. □

The consequences of Lemma 4.47 are somewhat disappointing. Asymptotically, the running time for a large product cannot be reduced by balancing the expression dag. However, if the maximum magnitude can be expected to stay reasonably small, then an improvement is possible.

Lemma 4.48. Let T_{list} be a list-like expression tree and let T_{bal} be a balanced expression tree, both consisting of n multiplications. Let $\hat{m}$ be the maximum exponent occurring during the evaluation of T_{list} and T_{bal}. Then the true cost of evaluating E_{list} or E_{bal} to accuracy z with the standard error distribution is given as

$$\mathrm{cost_t}(T_{list}) = \Theta(-nz + n^2\hat{m}) \quad \text{and} \quad \mathrm{cost_t}(T_{bal}) = \Theta(-nz + n\log(n)\hat{m})$$

Proof. The required accuracy for a node v_j on layer j is bounded by $-j\,(1+\hat{m})-1$ in the list-like case and by $-j\,(2+\hat{m})-1$ in the balanced case. Similar to the proof of Lemma 4.47, we get

$$\mathrm{cost_t}(T_{list}) \leq \sum_{j=0}^{n-1} (-z + j\,(1+\hat{m}) + 1 + \hat{m}) = O(-nz + n^2\hat{m}) \tag{4.16}$$

for the list-like operator tree and, with $k = \lceil \log(n+1) \rceil$,

$$\mathrm{cost_t}(T_{bal}) \leq \sum_{j=0}^{k-1} 2^j\,(-z + j\,(2+\hat{m}) + 1 + \hat{m}) = O(-nz + n\log(n)\hat{m})$$

for the balanced version, as well as matching lower bounds. □

Note that the quadratic term in (4.16) is not caused by the variable cost of the expression dag and therefore does not depend on the error distribution. By balancing error bounds as in Section 4.2, this term cannot be reduced. On the contrary, AM-Balancing is able to significantly reduce the true cost of the evaluation. Its applicability, however, is very limited. It can be extended by allowing semi-invasive operations.

Semi-Invasive Restructuring Methods

While asymptotically similar, the actual running time of the individual bigfloat operations differs significantly. Divisions are more expensive than square roots, square roots are more expensive than multiplications, and multiplications are far more expensive than additions or subtractions (cf. Table 2.3, page 16). Semi-invasive restructuring methods can be expected to improve the running time if they improve the structure and the restructured operator tree uses the same or less expensive operations than the original one. Furthermore, the running time can be improved by reducing the number of operations. Potential techniques for semi-invasive restructuring methods include:

Eliminating subsequent negations Whenever two subsequent negations occur inside an operator tree, they can be completely eliminated. The cost of the negation itself is negligible, but for each negation node an additional copy of the approximation must be maintained. Furthermore, eliminating negations potentially creates a structure that is better suited for other restructuring methods such as AM-Balancing.

Rising negations In multiplications or divisions, a negation of one of the operands can be eliminated by negating the result. A subsequent application of this rule can lead to subsequent negations which then can be eliminated. Rising negations may also improve the results of other restructuring methods if restructurable subtrees are separated by negations.

Replacing Subtractions Subtractions can be replaced by an addition and a negation with the benefit of increased associativity. The created addition node can then be used in methods like AM-Balancing. If negations are further propagated to the leaves, subsequent subtractions can be transformed into subsequent additions. Note that this method is not necessarily semi-invasive by definition, since additional negations are created. We list the method as semi-invasive because negations are cheap enough to be compensated by even a small improvement in structure.

Replacing subsequent divisions Subsequent divisions can be transformed into a series of multiplications followed by a single division. Since multiplications are cheaper than divisions, this usually leads to an improved running time. Afterwards the multiplications can be balanced through AM-Balancing, leading to an improvement in structure. This method also applies if there is a mixture of divisions and multiplications.

Combining subsequent roots Subsequent root operations can be replaced by a single root operation of higher degree. While computing a root of higher degree is more expensive than computing a root of small degree, the additional cost is compensated by the reduction in the number of operator nodes.

The proposed semi-invasive techniques can be expected to work reasonably well in their specific context, similar to AM-Balancing. If these strategies are combined, they can be applied to many special cases during an evaluation. Nevertheless, they still cover only a very small fraction of all existing expressions. For a more general approach, accepting a certain degree of invasiveness is inevitable.

4.3.3 Brent Restructuring

If the number of operations that need to be executed is allowed to increase, more general expressions can be effectively restructured. Richard Brent has shown that there is a restructuring algorithm achieving logarithmic depth for each operator tree consisting of all basic operations except for roots [Bre74]. We call this method *Brent Restructuring*. In contrast to the previous methods, Brent Restructuring is invasive. In this section we give a basic description of the algorithm and a short proof that it indeed always leads to logarithmic depth. Afterwards, we quantify the impact of its invasiveness. We first construct a sequence of operator trees, where Brent Restructuring always increases the size of the trees by factor four. Afterwards we prove, partly computer-assisted, that this factor is maximal.

Brent Restructuring

The Basic Algorithm

As before, we use the concepts of an expression and an expression dag interchangeably (cf. Remark 2.9). In this section, we extensively abuse this convention in order to reduce the technicalities and in the hope of creating more intuitively understandable statements. In particular, we use statements of the form $E = E_1 \circ E_2$ to express that E is an expression dag with operator $\circ$ (in its root node) and two children E_1 and E_2. The restructuring algorithm recursively creates normalized subexpressions with logarithmic depth, as specified in the following definition.

Definition 4.49. Let A, B, C, D, F, G be division-free expressions, let X be an expression and let E, E_X be expressions of the form

$$E = \frac{F}{G} \quad \text{and} \quad E_X = \frac{AX + B}{CX + D}$$

Divide-Structure
DAM-Structure
components
balanced

Then we call E a *Divide-Structure* and we call E_X a *Divide-Add-Multiply (DAM)-Structure* for X. The subexpressions A, B, C, D, F, G are called the *components* of E and E_X, respectively. We call E or E_X *balanced* if all of their components have a logarithmic depth.

The definition of a balanced Divide- or DAM-Structure is deliberately held a little vague since the actual constants involved differ slightly depending on the context. The introduced structures are used as containers for their respective components. Consequently, whenever we operate on either of these structures, we are primarily interested in the properties of their components. We introduce notations for their depth and their number of operators.

$\mathcal{DS}$
$\mathcal{DAM}(X)$

Definition 4.50. We denote the set of all Divide-Structures as $\mathcal{DS}$ and for an expression X we denote the set of DAM-Structures for X as $\mathcal{DAM}(X)$. Let E be a Divide-Structure with components $\mathcal{C}(E) = \{F, G\}$ or a DAM-Structure with components $\mathcal{C}(E) = \{A, B, C, D\}$. We set

$\mathrm{depth}_{\mathrm{c}}(E)$, $|E|_{\bullet}$

$$\mathrm{depth}_{\mathrm{c}}(E) = \max_{K \in \mathcal{C}(E)} (\mathrm{depth}(K)) \quad \text{as well as} \quad |E|_{\bullet} = \sum_{K \in \mathcal{C}(E)} |K|_{\circ}$$

component depth

The number $|E|_{\bullet}$ represents the number of operators of (the components of) E. We call $\mathrm{depth}_{\mathrm{c}}(E)$ the *component depth* or, if clear from the context, the *depth* of E.

$|E|_\iota$

In addition to the previous definition, we extend the notation for the number of operands of an operator dag to Divide- and DAM-Structures and denote the number of operands in the components of a respective structure E by $|E|_\iota$. If we use this notation, we generally ignore components of value zero or one that do not contain any nodes. These components are exclusively used for intermediate steps during the algorithm and never lead to an increase in the number of operands of the final result. Note that, while the number of operands of a Divide-Structure is equal to the number of operands of the underlying expression dag, the notation excludes the number of operands of X for a DAM-Structure $E \in \mathcal{DAM}(X)$.

The basic idea behind Brent's restructuring algorithm is to split the operator tree at half of its size into three parts, balance those parts recursively and put them together afterwards such that the bottom part of the expression has only a constant distance to the root (cf. Figure 4.11). The core of the restructuring algorithm is formed by the two functions `compress` and `raise` shown in Algorithm 6. The function `compress` takes an arbitrary operator tree T and returns a balanced Divide-Structure. The function `raise` takes an expression T and a subexpression X and returns a balanced DAM-Structure for X. It therefore effectively raises X up to a constant distance to the root of T. Note that

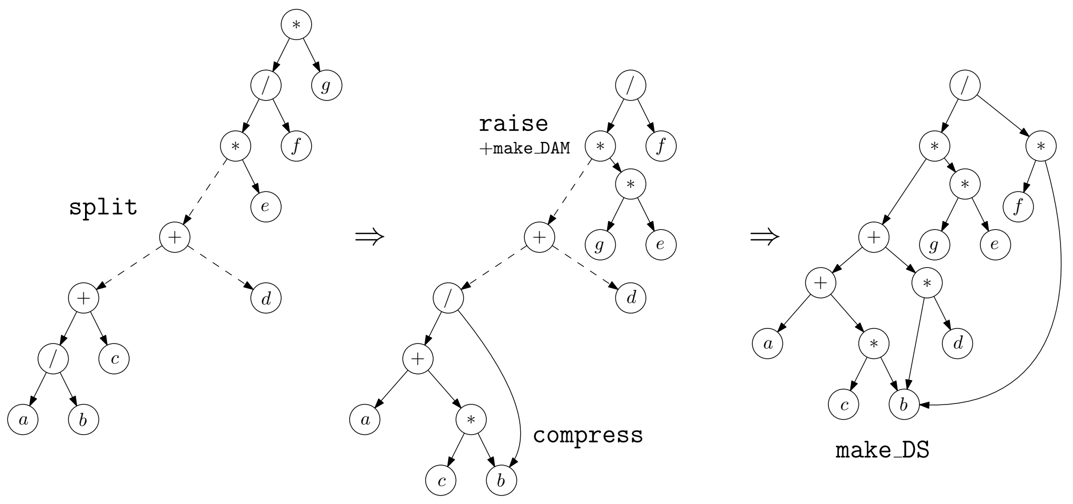


Figure 4.11: The basic recursive steps in Brent Restructuring. An expression is split into three parts with at most half of its size. The lower parts are recursively balanced and the split node is risen toward the root of the expression, thereby converting the expression to a balanced DAM-Structure for the subexpression at the split node. Afterwards, the three parts are combined into a balanced Divide-Structure.

Algorithm 6: The functions `compress` and `raise`.

```
1  **Function** compress($T$):
2      **if** $|T|_l > 1$ **then**
3          $X$ = split($T$,$\frac{1}{2}|T|_l$);
4          let $X_1, X_2$ be the children of $X$;
5          $X_1$ = compress($X_1$); $X_2$ = compress($X_2$); $T_X$ = raise($T$,$X$);
6          **return** make_DS($T_X$,$X$);
7      **end**
8
9  **Function** raise($T$,$X$):
10     **if** $T \neq X$ **then**
11         $Y$ = split($T$,$\frac{1}{2}(|T|_l + |X|_l)$);
12         let $Y_1, Y_2$ be the children of $Y$, such that $Y_1$ contains $X$;
13         $Y_1$ = raise($Y_1$,$X$); $Y_2$ = compress($Y_2$); $T_Y$ = raise($T$,$Y$);
14         **return** make_DAM($T_Y$,$Y$,$X$);
15     **end**
```

the result of `compress` is not necessarily a tree anymore but may introduce new operator nodes on common subexpressions. The `split` function used in line 3 and line 11 returns the minimal subexpression that contains at least half of the operands in T and is not an operand by itself. Its conceptual implementation is shown in Algorithm 7.

In both `compress` and `raise`, the three parts of the expression created by the split

Algorithm 7: The `split` function.

```
1 Function split($X$,$w$):
2     let $X_1, X_2$ be the children of $X$ with $|X_1|_l \geq |X_2|_l$;
3     if $X_1$ is not an operand and $|X_1|_l \geq w$ then
4         return split($X_1$,$w$);
5     else
6         return $X$;
7     end
```

are fused together at the end into a single Divide-Structure or a single DAM-Structure, respectively. In line 6 of Algorithm 6, the main part of the expression is a DAM-Structure of the form $E_X = (AX + B)/(CX + D)$, while X is of the form $F_1/G_1 \circ F_2/G_2$. The function `make_DS` creates an expression F_X/G_X for X according to Table 4.12 and then substitutes X into E_X, creating a new Divide-Structure F/G by setting $F = AF_X + BG_X$ and $G = CF_X + DG_X$.

	$X_1 + X_2$	$X_1 - X_2$	$X_1 \cdot X_2$	X_1/X_2
F_X	$F_1G_2 + F_2G_1$	$F_1G_2 - F_2G_1$	F_1F_2	F_1G_2
G_X	G_1G_2	G_1G_2	G_1G_2	G_1F_2

Table 4.12: The components of the Divide-Structure F_X/G_X for X created by `make_DS` if $X = X_1 \circ X_2 = F_1/G_1 \circ F_2/G_2$ with operator $\circ \in \{+, -, \cdot, /\}$.

The substitution function `make_DAM` used in `raise` in line 14 is more complicated. It creates a DAM-Structure for X by substituting Y, which is an expression of the form

$$Y = Y_X \circ Y_C \quad \text{or} \quad Y = Y_C \circ Y_X$$

with the operands

$$Y_X = \frac{A_X X + B_X}{C_X X + D_X} \text{ and } Y_C = \frac{F}{G}$$

into the existing DAM-Structure

$$E_Y = \frac{A_Y Y + B_Y}{C_Y Y + D_Y}$$

The structure of the new components A, B, C, D depends on the operator $\circ$ and on the order of the operands Y_X, Y_C. In Table 4.13, the structure of the components is shown for the case in which $\circ$ is an addition or a multiplication. Subtraction and division can be reduced to addition and multiplication by negating or, respectively, inverting the second operand. Since both addition and multiplication are associative, the order of Y_X, Y_C can then be changed to match the order assumed in Table 4.13.

Brent shows in his work that the depth of the expression after restructuring is logarithmic in the number of operands. We use a simplified variant of his proof to show that our implementation results in logarithmic depth.

	$Y = Y_X + Y_C$	$Y = Y_X \cdot Y_C$
A	$A_Y(A_XG + C_XF) + B_Y(C_XG)$	$A_Y(A_XF) + B_Y(C_XG)$
B	$A_Y(B_XG + D_XF) + B_Y(D_XG)$	$A_Y(B_XF) + B_Y(D_XG)$
C	$C_Y(A_XG + C_XF) + D_Y(C_XG)$	$C_Y(A_XF) + D_Y(C_XG)$
D	$C_Y(B_XG + D_XF) + D_Y(D_XG)$	$C_Y(B_XF) + D_Y(D_XG)$

Table 4.13: The components of the DAM-Structure $(AX+B)/(CX+D)$ created with the function `make_DAM` by substituting Y into the DAM-Structure E_Y.

Theorem 4.51 (Based on [Bre74]). Let T be an operator tree and T' be the operator tree created through Brent Restructuring. Then the depth of T' is at most $4\lceil\log(|T|_l)\rceil$.

Proof. We prove by induction that for each Divide-Structure $E \in \mathcal{DS}$ with $|E|_l \geq 2$ and DAM-Structure $E_X \in \mathcal{DAM}(X)$ with $|E_X|_l \geq 2$ created during the algorithm we have

$$\operatorname{depth_c}(E) \leq 4\lceil\log(|E|_l)\rceil - 3 \quad \text{and} \quad \operatorname{depth_c}(E_X) \leq 4\lceil\log(|E_X|_l)\rceil - 3$$

If $|E|_l = 1$, then $\operatorname{depth_c}(E) = 0$. If $|E|_l = 2$, then we need at most one operation to compute its components, so $\operatorname{depth_c}(E) \leq 1 = 4\log(|E|_l) - 3$. Similarly, if $|E_X|_l = 0$ or $|E_X|_l = 1$, then $\operatorname{depth_c}(E_X) = 0$ and if $|E_X|_l = 2$, then $\operatorname{depth_c}(E_X) \leq 1 = 4\log(|E_X|_l) - 3$. For the induction step, let $E_X \in \mathcal{DAM}(X)$ with $|E_X|_l = k \geq 3$. Then there are expressions E_Y, Y, Y_X and Y_C such that

1. $E_X = \texttt{make\_DAM}(E_Y, Y, X)$
2. $E_Y \in \mathcal{DAM}(Y)$ with $Y = Y_X \circ Y_C$ or $Y = Y_C \circ Y_X$ and $|E_Y|_l \leq \frac{k}{2}$
3. $Y_X \in \mathcal{DAM}(X)$ with $|Y_X|_l < \frac{k}{2}$
4. $Y_C \in \mathcal{DS}$ with $|Y_C|_l < \frac{k}{2}$

Since each expression for the components of E_X has a depth of at most 4 in the components of E_Y, Y_X, Y_C (cf. Table 4.13), we get

$$\begin{aligned} \operatorname{depth_c}(E_X) &\leq \max(\operatorname{depth_c}(E_Y), \operatorname{depth_c}(Y_X), \operatorname{depth_c}(Y_C)) + 4 \\ &\leq 4\lceil\log(k/2)\rceil - 3 + 4 \\ &= 4\lceil\log(k)\rceil - 3 \end{aligned}$$

Now let $E \in \mathcal{DS}$ with $|E|_l = k \geq 3$. Then there are E_Y, Y_1 and Y_2 with $|E_Y|_l, |Y_1|_l, |Y_2|_l \leq k/2$ such that $E = \texttt{make\_DS}(E_Y, Y)$ as well as $E_Y \in \mathcal{DAM}(Y)$ with $Y = Y_1 \circ Y_2$ and $Y_1, Y_2 \in \mathcal{DS}$. As before, each of the components of E has a depth of at most 4 in the components of E, Y_1, Y_2. Therefore,

$$\begin{aligned} \operatorname{depth_c}(E) &\leq \max(\operatorname{depth_c}(E_Y), \operatorname{depth_c}(Y_1), \operatorname{depth_c}(Y_2)) + 4 \\ &\leq 4\lceil\log(k/2)\rceil - 3 + 4 \\ &= 4\lceil\log(k)\rceil - 3 \end{aligned}$$

Finally, there is a Divide-Structure $E_T \in \mathcal{DS}$ with $E_T = \texttt{compress}(T)$ and $|E_T|_l = |T|_l$. Therefore, $\operatorname{depth}(T') = \operatorname{depth_c}(E_T) + 1 \leq 4\lceil\log(|T|_l)\rceil$. □

Invasiveness of Brent Restructuring

Brent Restructuring is guaranteed to reach a near-optimal expression depth for significantly more general expressions than AM-Balancing. Nevertheless it can lead to significantly worse structures. We define an expression dag as shown in Figure 4.14. Let E^k_{div} be an expression dag with $n = 2k$ operands $x_1, ..., x_{2k}$ defined by

$$E^k_{div} = \begin{cases} \frac{x_1}{x_2} & \text{if } k = 1 \\ E^{k-1}_{div} + \frac{x_{2k-1}}{x_{2k}} & \text{else} \end{cases} \tag{4.17}$$

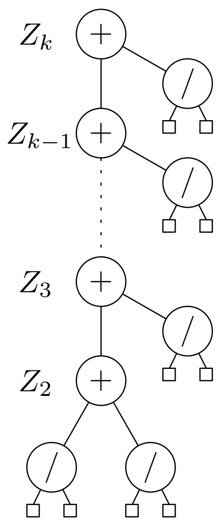


Figure 4.14: The graph E^k_{div}.

Then AM-Balancing leads to an optimal expression dag of height $\lceil \log(n) \rceil$ by balancing the tree of additions on top of all the division nodes. Brent Restructuring, on the other hand, incorporates the divisions in the leaves and, in this process, creates several additional multiplication nodes.

Lemma 4.52. Let $k = 2^m - 1$ with $m \geq 3$, let T be the unique maximal operator tree of E^k_{div} and let T' be the operator tree created from T by Brent Restructuring. Then the number of operators used in the new operator dag is $|T'|_\circ = 2.5(|T|_\circ - 1) - 2\log(|T|_\circ + 3)$.

Proof. First note that whenever a `split` operation happens, the split node is an addition node. Therefore, `raise` is always called on addition nodes. Furthermore, each `compress` operation on a division node returns the same division in form of a Divide-Structure with operator number zero. For $E_X = \texttt{make\_DAM}(E_Y, Y, X)$, only two possible cases occur during the restructuring.

Case 1: $E_Y = Y$ and $Y = Y_X + Y_C$ and $Y_X = X$ and $Y_C = F/G$

Case 2: $E_Y = (AY + B)/D$ and $Y = Y_X + Y_C$ and $Y_X = (aX + b)/d$ and $Y_C = F/G$

Both cases result in a DAM-Structure $E_X \in \mathcal{DAM}(X)$ of the form $(AX + B)/D$. In Case 1, we do not need any operations to calculate the components of E_X. In Case 2, we need 10 operations for their calculation (cf. Table 4.13). Let $Z_2, ..., Z_k = E^k_{div}$ be the subexpressions of E^k_{div} rooted at the addition nodes, starting from the leaves. A call to $R_0 = \texttt{raise}(Z_{j+1}, Z_j)$ results in Case 1. A call to $R = \texttt{raise}(Z_{j+2^m-1}, Z_j)$ with $m \geq 2$ consists of one call to `compress` for the division node at Z_{j+2^m-1} as well as calls to $R_1 = \texttt{raise}(Z_{j+2^m-1}, Z_{j+2^{m-1}})$ and $R_2 = \texttt{raise}(Z_{j+2^{m-1}-1}, Z_j)$. By induction, it follows that Case 2 applies and since $|R_0|_\bullet = 10(2^0 - 1) = 0$, the number of operators in the resulting DAM-Structure $R \in \mathcal{DAM}(Z_j)$ is given as

$$|R|_\bullet = |R_1|_\bullet + |R_2|_\bullet + 10 = 10(2^{m-2} - 1) + 10(2^{m-2} - 1) + 10 = 10(2^{m-1} - 1)$$

For `make_DS` we get only one case, leading to a Divide-Structure $C = F/G$.

Case A: $E = (AX + B)/D$ and $X = X_1 + X_2$ and $X_1 = F_1/G_1$ and $X_2 = F_2/G_2$

Calling `compress` on Z_3 requires a call to $\texttt{raise}(Z_3, Z_2)$ and two calls to `compress` for the children of Z_2. This results in Case A in which we need $|C|_\bullet = 8$ operators to represent its components (cf. Table 4.12). A call to $C = \texttt{compress}(Z_{2^m-1})$ with $m \geq 3$ consists of calls to $R = \texttt{raise}(Z_{2^m-1}, Z_{2^{m-1}})$, to $C_1 = \texttt{compress}(Z_{2^{m-1}-1})$ and to `compress` for the division node at $Z_{2^{m-1}}$. As mentioned before, for $m = 2$ we have $|C|_\bullet = 8 = 10(2^{m-1} - 1) - 2(m-1)$. For each $m \geq 3$, we are in Case A as well. By induction we get

$$|C|_\bullet = |R|_\bullet + |C_1|_\bullet + 8 = 10(2^{m-2}-1) + 10(2^{m-2}-1) - 2(m-2) + 8 = 10(2^{m-1}-1) - 2(m-1)$$

In order to restructure T, `compress` is called and the result is combined by applying one division on the resulting Divide-Structure. Therefore, $|T'|_\circ = 5(k-1) - 2\log(k+1) + 3$. Since $|T|_\circ = 2k - 1$, we have proven the lemma. □

Lemma 4.52 proves its bound only for $k = 2^m - 1, m \geq 3$. With a bit more effort it can be shown, that for all $k \geq 1$ we need $5k - O(\log k)$ operators. Since both AM-Balancing and Brent Restructuring lead to a logarithmic depth for E^k_{div}, we can expect the cost of the evaluation after Brent Restructuring to be at least 2.5 times the cost of the evaluation with the non-invasive method. We formalize this concept in the following definition:

Definition 4.53. Let R be a restructuring algorithm with domain $\mathcal{D}$ and let $\mathcal{S}$ be the set of all sequences $(T_n)_{n\in\mathbb{N}}$ of operator trees $T_n \in \mathcal{D}$ such that $|T_{n+k}|_l > |T_n|_l$ for all $k > 0$ and $|R(T_n)|_\circ / |T_n|_\circ$ converges in $\mathbb{R} \cup \{\infty\}$ for $n \to \infty$. Then

$$\alpha = \sup \left\{ \lim_{n\to\infty} \frac{|R(T_n)|_\circ}{|T_n|_\circ} \;\middle|\; (T_n)_{n\in\mathbb{N}} \in \mathcal{S} \right\}$$

is called the *degree of invasiveness* of R.

degree of invasiveness

Note that the degree of invasiveness is always defined since for every sequence of trees the range of $|R(T_n)|_\circ / |T_n|_\circ$ is bounded by $[0, \infty]$ and therefore, by the Bolzano-Weierstraß theorem, it contains a converging subsequence.

A Lower Bound on the Degree

Obviously, $\alpha = 1$ for non-invasive and semi-invasive restructuring methods. Let α_B be the degree of invasiveness of Brent Restructuring. From Lemma 4.52 we already know that $\alpha_B \geq 2.5$. In Brent's original work, he shows that $\alpha_B \leq 10$. In this work, we show that Brent Restructuring has a degree of invasiveness of $\alpha_B = 4$. We split the proof of this statement into several parts. First, we give a construction for the lower bound. Let E^k_{max} be an expression dag with $n = 5k + 1$ operands $x_0, ..., x_{5k}$ defined by

$$E^k_{max} = \begin{cases} x_0 & \text{if } k = 0 \\ \frac{x_{5k-1}}{x_{5k}} + \left(x_{5k-2} + \frac{x_{5k-3}}{x_{5k-4} + E^{k-1}_{max}} \right) & \text{else} \end{cases} \tag{4.18}$$

The path from the root to x_0 consists of k blocks of two additions, one division and another addition. We denote subtrees rooted at the $4k$ operator nodes along this path from top to bottom as $Z_{4k}, ..., Z_1$. So the root of Z_i contains a division node for $i = 4j-2$, $1 \leq j \leq k$ and an addition node otherwise (cf. Figure 4.15). We first deduce a property that helps us to control where the splits of `compress` and `raise` happen during the algorithm.

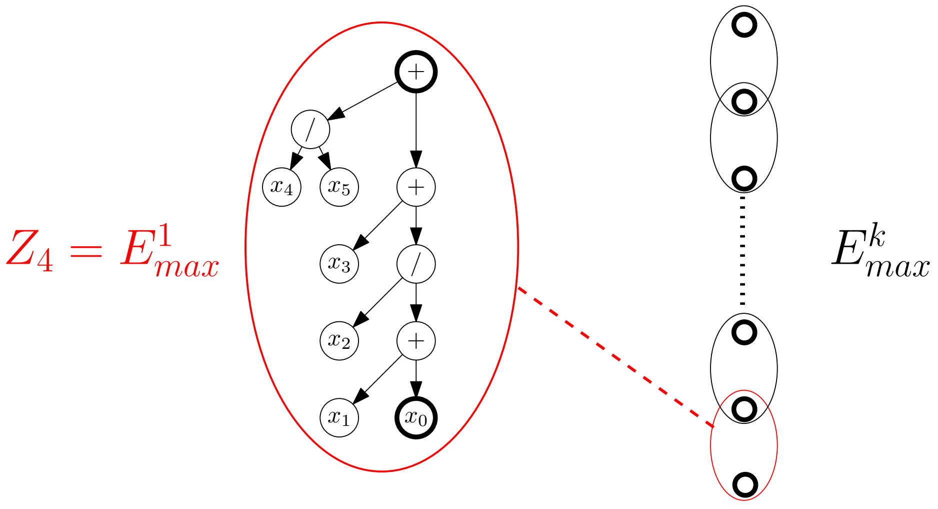


Figure 4.15: The expression dag E^k_{max}, formed by k equally structured building blocks.

Lemma 4.54. Let $m \geq 3$ and consider E^k_{max} for $k \geq 2$. Then each call to `compress`(Z_{2^m}) and each call to `compress`(Z_{2^m-1}) splits at $Z_{2^{m-1}}$. Furthermore, for $j \geq 0$ each call to `raise`(Z_{4j+2^m}, Z_{4j}) and each call to `raise`(Z_{4j+2^m-1}, Z_{4j}) splits at $Z_{4j+2^{m-1}}$.

Proof. Each of the basic building blocks of E^k_{max}, except for the first one, consists of five operands. The respective numbers of operands in the subexpressions $Z_1, ..., Z_{4k}$ are given as

$$|Z_i|_l = \left\lfloor \frac{5i}{4} \right\rfloor + 1 \tag{4.19}$$

Note that for every fourth subexpression, the number of operands increases by two due to the additional division. The split of `compress`(E) happens at the lowest node in E at which the number of operands is at least half of $|E|_l$ (cf. line 3 of Algorithm 6). For `compress`, we have $|Z_{2^m}|_l = 5 \cdot 2^{m-2} + 1$ and $|Z_{2^m-1}|_l = 5 \cdot 2^{m-2} - 1$. Since $|Z_{2^{m-1}}|_l = 5 \cdot 2^{m-3} + 1$, whereas its child nodes have at most $5 \cdot 2^{m-3} - 1$ operands, in both cases $Z_{2^{m-1}}$ is the split node of the `compress` operation. The split of `raise`(E, X) happens at the lowest node in the subexpression where the number of operands is at least half the number of operands of E and X combined (cf. line 11 in Algorithm 6). Since

$$\left\lceil \frac{|Z_{4j+2^m}|_l + |Z_{4j}|_l}{2} \right\rceil = 5j + 5 \cdot 2^{m-3} + 1 \quad \text{and} \quad \left\lceil \frac{|Z_{4j+2^m-1}|_l + |Z_{4j}|_l}{2} \right\rceil = 5j + 5 \cdot 2^{m-3}$$

in both cases the split happens at $Z_{4j+2^{m-1}}$ with $|Z_{4j+2^{m-1}}|_l = 5j + 5 \cdot 2^{m-3} + 1$. □

With Lemma 4.54, we know where splits happen if we choose powers of two for the number of building blocks. The functions `make_DS` and `make_DAM` generate a maximum amount of additional operator nodes when all components of the respective substructures are relevant for the creation of the new structure.

Definition 4.55. Let $E_X \in \mathcal{DAM}(X)$ be a DAM-Structure of the form

$$E_X = \frac{AX + B}{CX + D}$$

with $|A|_l, |B|_l, |C|_l, |D|_l \geq 1$. Then E_X is called *complete*. Likewise, we call a Divide-Structure $E \in \mathcal{DS}$ with $E = F/G$ *complete*, if $|F|_l \geq 1$ and $|G|_l \geq 1$. complete

Each call to `make_DAM` produces 20 operators if the involved substructures are complete. We show that two building blocks are sufficient to generate a complete DAM-Structure and that during a call to `raise` where the raised node is a power of two buildings blocks away from its destination, on average, each building block generates 20 operators. Note that each building block contributes 2^2 operator nodes to the longest path.

Lemma 4.56. For the expression dag E^k_{max} with $k \geq 2$ let $E_1, E_2 \in \mathcal{DAM}(Z_{4j})$ be DAM-Structures created by $E_1 = \texttt{raise}(Z_{4j+2^m-1}, Z_{4j})$ and $E_2 = \texttt{raise}(Z_{4j+2^m}, Z_{4j})$ for $m \geq 3$ and $j \geq 0$. Then the following two statements hold:

1. E_1 is complete and $|E_1|_\bullet = 20(2^{m-2} - 1)$.
2. E_2 is complete and $|E_2|_\bullet = 20(2^{m-2} - 1) + 8$.

Proof. All calls to `compress` needed for the creation of E_1 and E_2 operate either on a single operand or on a division between two operands, leading to Divide-Structures of the form F or F/G where all components are operands and therefore the number of required operators is 0. For `raise`, we get ten different non-trivial cases. The seven base cases are depicted in Table 4.16. Case 8 and Case 9 are depicted in Figure 4.17 and described in the following. In the tenth case, each substructure is complete and the operation at the split node is an addition, leading to 20 additional operations (cf. Table 4.13). We first show that $\texttt{raise}(Z_{4j+2^m-1}, Z_{4j})$ with $m \geq 3$ leads to a complete DAM-Structure with $20(2^{m-2}-1)$ operators. Due to Lemma 4.54, the split occurs at the root of $Z_{4j+2^{m-1}}$. Let $m = 3$, then we are in Case 9 as depicted in Figure 4.17 and both subsequent recursive calls eventually land in Case 6. For computing $\texttt{make\_DAM}(Z_{4j+7}, Z_{4j+4}, Z_{4j})$ we then get

$$E_Y = \frac{AY + B}{Y + D}, \quad Y = \frac{F}{G} + Y_X, \quad Y_X = \frac{aX + b}{X + d}$$

and therefore

$$E_X = \frac{(A(aG + F) + BG)X + (A(bG + dF) + B(dG))}{((aG + F) + DG)X + ((bG + dF) + D(dG))}$$

This step produces 16 operators. To compute the number of operators required for the components, only the final cases are relevant, adding another 4 operators. Therefore, E_X

	E_Y	Y_C	$\circ$	Y_X	E_X	$\lvert E_X\rvert_\bullet$
Case 1	Y	F	$+$	X	$X+F$	$0+\lvert F\rvert_\circ$
Case 2	Y	F	$/$	X	F/X	$0+\lvert F\rvert_\circ$
Case 3	Y	F/G	$+$	X	$(GX+F)/G$	$0+\lvert F,G\rvert_\circ$
Case 4	B/Y	F	$+$	X	$B/(X+F)$	$0+\lvert B,F\rvert_\circ$
Case 5	$Y+B$	F	$/$	X	$(BX+F)/X$	$0+\lvert B,F\rvert_\circ$
Case 6	$Y+B$	F	$/$	$X+b$	$\frac{BX+(F+Bb)}{X+b}$	$2+\lvert B,F,b\rvert_\circ$
Case 7	$\frac{AY+B}{D}$	F	$+$	$\frac{b}{X+d}$	$\frac{(AF+b)X+(A(b+dF)+Bd)}{DX+Dd}$	$8+\lvert A,B,D,F,b,d\rvert_\circ$

Table 4.16: Substitution cases for the recursive `raise` calls in Lemma 4.56, where $E_X = \texttt{make\_DAM}(E_Y, Y, X)$ with $E_Y \in \mathcal{DAM}(Y)$ and $Y = Y_C \circ Y_X$. The notation $|\sigma_1,\sigma_2|_\circ$ is used as equivalent to $|\sigma_1|_\circ + |\sigma_2|_\circ$.

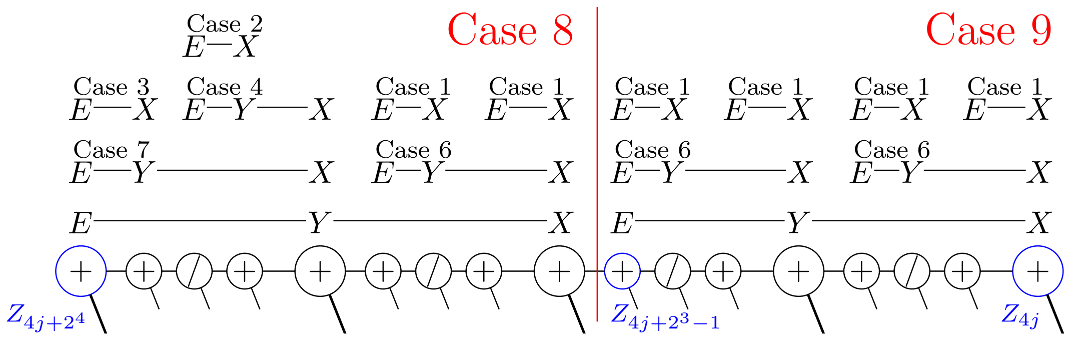


Figure 4.17: Case 8 and Case 9 and their subcases as they occur during calls to $\texttt{raise}(Z_{4j_1+2^3}, Z_{4j_1})$ and $\texttt{raise}(Z_{4j_2+2^3-1}, Z_{4j_2})$, respectively. For each (sub-)case the expression at the target node (E), the expression at the raised node (X) and the expression at the split node (Y) are marked. If no split node is marked, then $E = Y$.

is a complete DAM-Structure with $|E_X|_\bullet = 20$. For $m \geq 4$, by induction, the subsequent calls to $\texttt{raise}(Z_{4j+2^m-1}, Z_{4j+2^{m-1}})$ and $\texttt{raise}(Z_{4j+2^{m-1}}, Z_{4j})$ result in complete DAM-Structures with $20(2^{m-3}-1)$ operators. Since the second child node of $Z_{4j+2^{m-1}}$ is a division, we get

$$E_Y = \frac{AY+B}{CY+D}, \quad Y = \frac{F}{G} + Y_X, \quad Y_X = \frac{aX+b}{cX+d}$$

and therefore

$$E_X = \frac{(A(aG+cF)+B(cG))X + (A(bG+dF)+B(dG))}{(C(aG+cF)+D(cG))X + (C(bG+dF)+D(dG))}$$

Since this step produces 20 operators, we get

$$|E_X|_\bullet = 20(2^{m-3}-1) + 20(2^{m-3}-1) + 20 = 20(2^{m-2}-1)$$

We now show that $\texttt{raise}(Z_{4j+2^m}, Z_{4j})$ with $m \geq 3$ leads to a complete DAM-Structure with $20(2^{m-2}-1)+8$ operators. Again with Lemma 4.54, the root of $Z_{4j+2^{m-1}}$ is the split node of the `raise` operation. For $m = 3$, the subsequent recursive calls eventually land in Case 7 and Case 6. This is depicted as Case 8 in Figure 4.17. For computing `make_DAM`$(Z_{4j+8}, Z_{4j+4}, Z_{4j})$ we get

$$E_Y = \frac{AY + B}{CY + D}, \quad Y = \frac{F}{G} + Y_X, \quad Y_X = \frac{aX + b}{X + d}$$

and therefore

$$E_X = \frac{(A(aG + F) + BG)X + (A(bG + dF) + B(dG))}{(C(aG + F) + DG)X + (C(bG + dF) + D(dG))}$$

Obviously, the resulting DAM-Structure is complete. This step produces 18 operators. Another 10 operators are added to compute the components, resulting in $|E_X|_\bullet = 28$. Now, for $m \geq 4$ the subsequent calls are of the form `raise`$(Z_{4j+2^m}, Z_{4j+2^{m-1}})$ and `raise`$(Z_{4j+2^{m-1}}, Z_{4j})$. By induction and by the first statement of the lemma, both resulting structures are complete and, as before, we need 20 operators to represent the components of the result. Therefore,

$$|E_X|_\bullet = 20(2^{m-3} - 1) + 8 + 20(2^{m-3} - 1) + 20 = 20(2^{m-2} - 1) + 8 \qquad \square$$

With the results for `raise`, we can now show that each building block generates on average 20 operators during a call to `compress` if the number of building blocks is a power of two.

Lemma 4.57. For the expression dag E^k_{max} with $k \geq 2$ let $E \in \mathcal{DS}$ be created by $E =$ `compress`(Z_{2^m}) for $m \geq 5$. Then E is complete and the number of operators required to compute its components is $|E|_\bullet = 20(2^{m-2} - 1) - 10(m - 2) - 4$.

Proof. We first show that $E =$ `compress`(Z_{2^m-1}) leads to a complete Divide-Structure of size $20(2^{m-2} - 1) - 10(m - 2) + 4$. Let $m = 4$. Then by Lemma 4.54, `compress`(Z_{15}) splits at Z_8. With Lemma 4.56, `raise`(Z_{15}, Z_8) returns a complete DAM-Structure with 20 operators. We get three non-trivial cases A, B and C for `compress`, which are listed in Table 4.18. The computation for `compress`(Z_7) is shown in Figure 4.19. It returns a complete Divide-Structure with 14 operators. Since the second child of Z_8 is a division node, all three substructures are complete and `make_DAM`(Z_{15}) is in Case C. It follows that

	E_X	X_1	$\circ$	X_2	E	operators
Case A	$\frac{AX+B}{X}$	F_1	$+$	F_2	$\frac{A(F_1+F_2)+B}{F_1+F_2}$	3
Case B	$\frac{AX+B}{X+D}$	F_1/G_1	$+$	F_2/G_2	$\frac{A(F_1G_2+F_2G_1)+B(G_1G_2)}{(F_1G_2+F_2G_1)+D(G_1G_2)}$	9
Case C	$\frac{AX+B}{CX+D}$	F_1/G_1	$+$	F_2/G_2	$\frac{A(F_1G_2+F_2G_1)+B(G_1G_2)}{C(F_1G_2+F_2G_1)+D(G_1G_2)}$	10

Table 4.18: Substitution cases for recursive `compress` calls in an evaluation of E^k_{max}, where $E =$ `make_DS`(E_X, X) with $E_X \in \mathcal{DAM}(X)$ and $X = X_1 \circ X_2$. The last column lists the number of operators needed to create the components of E from the components of E_X, X_1 and X_2.

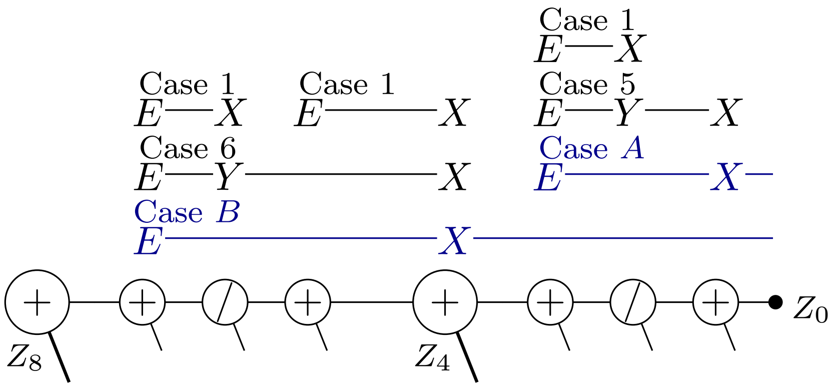


Figure 4.19: The cases occurring during a call to `compress`(Z_7). The two non-trivial `compress` calls are shown in blue. The expression at the target node is marked as E and the expression at the split node is marked as X. Subsequent raise cases are displayed as in Figure 4.17.

$|E|_\bullet = 20 + 14 + 10 = 44 = 20(2^{m-2} - 1) - 10(m-2) + 4$. For $m \geq 5$ the recursive calls are `raise`$(Z_{2^m-1}, Z_{2^{m-1}})$ and `compress`$(2^{m-1} - 1)$. By induction and by Lemma 4.56, both resulting structures are complete and we are again in Case C, therefore the number of operators for $E =$ `make_DS`$(Z_{2^m-1}, Z_{2^{m-1}})$ is given as

$$|E|_\bullet = 20(2^{m-3} - 1) + 20(2^{m-3} - 1) - 10(m-3) - 4 + 10 = 20(2^{m-2} - 1) - 10(m-2) + 4$$

Now, for $E =$ `compress`(Z_{2^m}) with $m \geq 5$ we get recursive calls to `raise`$(Z_{2^m}, Z_{2^{m-1}})$ and `compress`$(2^{m-1} - 1)$. With the above result and Lemma 4.56, we are again in Case C and we get

$$|E|_\bullet = 20(2^{m-3} - 1) + 8 + 20(2^{m-3} - 1) - 10(m-3) - 4 + 10 = 20(2^{m-2} - 1) - 10(m-2) - 4$$

□

Theorem 4.58. The degree of invasiveness of Brent Restructuring is at least 4.

Proof. Let $(T_n)_{n\in\mathbb{N}}$ be the sequence of operator trees with T_n being the unique maximal operator tree of E^k_{max} with $k = 2^n$. With Lemma 4.57, the number of operator nodes of T_n after restructuring is $20k - O(\log k)$ for $n \geq 5$. Since T_n consists of $5k$ operator nodes, the theorem follows. □

An Upper Bound on the Degree

To prove the upper bound we show that for large numbers of operands n each call to `raise` leads to a DAM-Structure with at most $4n - 12$ operators and each call to `compress` leads to a Divide-Structure with at most $4n - 8$ operators. We first show that these bounds lead to a valid induction step.

Lemma 4.59. Let $E_Y \in \mathcal{DAM}(Y)$ and $Y_X \in \mathcal{DAM}(X)$ and let $Y_1, Y_2 \in \mathcal{DS}$, such that

$$|E_Y|_\bullet \leq 4|E_Y|_\imath - 12 \;\text{ and }\; |Y_X|_\bullet \leq 4|Y_X|_\imath - 12$$

as well as

$$|Y_1|_\bullet \leq 4|Y_1|_l - 8 \text{ and } |Y_2|_\bullet \leq 4|Y_2|_l - 8$$

Then for $E_X = \texttt{make\_DAM}(E_Y, Y, X)$ with $Y = Y_X \circ Y_1$ or $Y = Y_1 \circ Y_X$ and for $E = \texttt{make\_DS}(E_Y, Y)$ with $Y = Y_1 \circ Y_2$ we get

$$|E_X|_\bullet \leq 4|E_X|_l - 12 \text{ and } |E|_\bullet \leq 4|E|_l - 8 \tag{4.20}$$

Proof. As depicted in Table 4.13, the components of E_X are represented by the components of the substructures with at most 20 operators. Therefore, we get

$$\begin{aligned}|E_X|_\bullet &\leq |E_Y|_\bullet + |Y_X|_\bullet + |Y_1|_\bullet + 20\\ &\leq 4|E_Y|_l - 12 + 4|Y_X|_l - 12 + 4|Y_1|_l - 8 + 20\\ &= 4(|E_Y|_l + |Y_X|_l + |Y_1|_l) - 12\end{aligned}$$

For representing the components of E, at most 10 operators are needed. This number splits into 4 operators for representing the components of Y (cf. Table 4.12) and 6 operators for representing $AY + B$ and $CY + D$. Therefore, the total number of operators is given as

$$\begin{aligned}|E|_\bullet &\leq |E_Y|_\bullet + |Y_1|_\bullet + |Y_2|_\bullet + 10\\ &\leq 4|E_Y|_l - 12 + 4|Y_1|_l - 8 + 4|Y_2|_l - 8 + 10\\ &< 4(|E_Y|_l + |Y_1|_l + |Y_2|_l) - 8\end{aligned}$$

□

We call (4.20) the *operator bound property* for a Divide- or DAM-Structure. We prove the base case of the induction with a computer-assisted approach. For small numbers of operands, exhaustive search is applied to determine the maximum number of operator nodes after restructuring. We give a sketch to the tree generation algorithm. Every ordered binary tree with n leaves is isomorphic to a sequence of natural numbers

op. bound property

Algorithm 8: Algorithm for generating a unique expression tree with additions, multiplications and divisions from each sequence of $n - 1$ natural numbers $a_0, .., a_{n-2}$ with $0 \leq a_i \leq 3(n - i) - 1$.

```
Function generate_tree([$a_0, ..., a_{n-2}$]):
    let [$x_0, .., x_{n-1}$] be operands;
    for $i = 0$ to $n - 2$ do
        $j = \lfloor i/3 \rfloor$; $k = i \mod 3$;
        $l_{a_j}$ = make_node ($l_{a_j}$,$l_{a_j+1}$,$\circ_k$);
        remove $l_{a_j+1}$;
    end
    return $l_1$
```

$a_0, ..., a_{n-2}$ with $0 \leq a_i \leq n - i - 1$. A tree can be generated by such a sequence by starting with n ordered leaves and subsequently contracting the a_i-th node with its neighbor. For generating all possible operator trees, we additionally incorporate the number of different operations for each node into the sequence. We use every combination of additions, multiplications and divisions. We do not use subtractions, since they lead to the same structure as additions aside from occasionally adding a negation, and we ignore roots, since they cannot be processed by Brent Restructuring. The operations are encoded by extending the range of each a_i to $0 \leq a_i \leq 3(n - i) - 1$ and then treating every first, second and third node as addition, multiplication or division as depicted in Algorithm 8. We use exhaustive search to find the maximum number of operator nodes

	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
`compress`	–	0	1	2	4	6	9	11	14
`raise`	0	0	1	3	5	8	12	16	20

Table 4.20: Maximum number of operators for a Divide-Structure generated by `compress`(E) and a DAM-Structure generated by `raise`(E, X) depending on the number of operands $n = |E|_\iota$ and $n = |E_X|_\iota$, respectively. The numbers were determined by an exhaustive search over all trees containing additions, multiplications and divisions.

after executing both `compress` and `raise`. In the case of `raise`, we generate trees over $n + 1$ operands and repeat the process $n + 1$ times while marking one of the operands as the node that should be raised. Table 4.20 shows the maximum number of operators a Divide-Structure generated by `compress` and a DAM-Structure generated by `raise` can have for $n \leq 8$ operands (not counting the marked node in the case of `raise`). For each case, exemplary operator trees for which a maximum number of new operator nodes are generated are shown in Figure 4.21 and Figure 4.22.

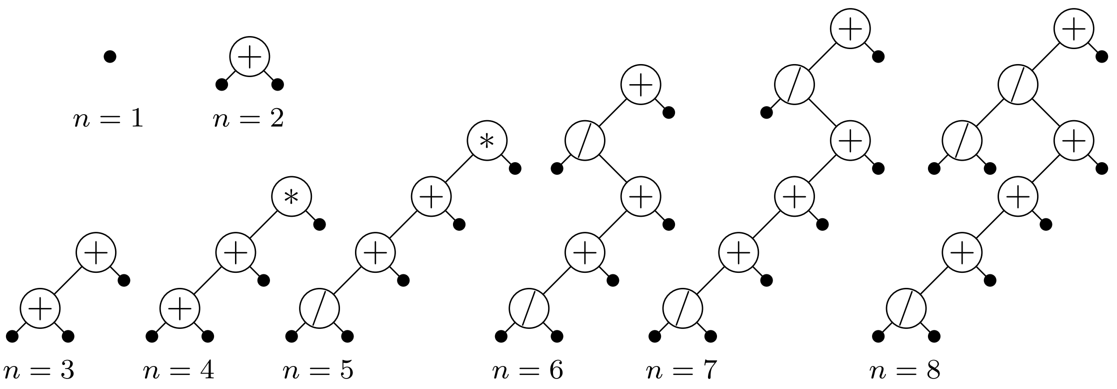


Figure 4.21: Operator trees with $n \leq 8$ operands for which `compress` creates a maximum number of new operator nodes.

The operator bound property (4.20) is fulfilled for $n \geq 3$ for `compress` and for $n \geq 5$ for `raise`. Smaller values of n must be addressed separately in the induction step. For this, we use the fact that the maximal structures for small n are usually not complete and therefore do not lead to a maximum increase in the number of operators.

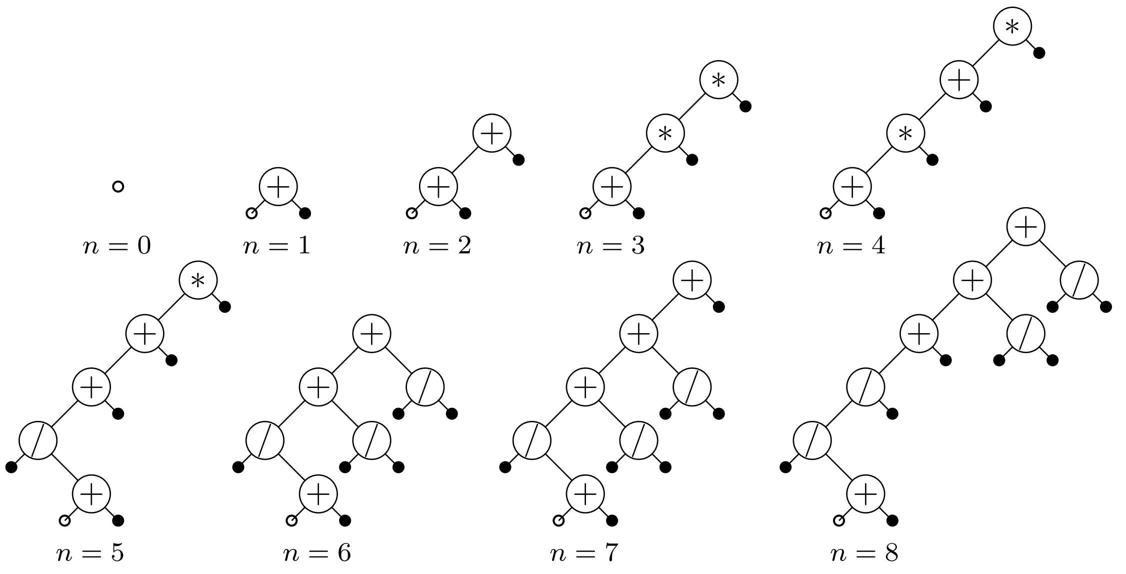


Figure 4.22: Segments of operator trees with $n \leq 8$ operands for which `raise` creates a maximum number of new operator nodes. The node containing the subexpression that should be raised is depicted as an empty circle.

Definition 4.60. For $E_Y \in \mathcal{DAM}(Y)$, $Y_X \in \mathcal{DAM}(X)$ and $Y_C \in \mathcal{DS}$ let

$$\boldsymbol{Y} = \{Y_X \circ Y_C, Y_C \circ Y_X \mid \circ \in \{+, -, \cdot, /\}\}$$

be the set of all possible expressions at a respective split node. Then for $E_X(Y) =$ `make_DAM`(E_Y, Y, X) we call

$$\delta^\circ(E_Y, Y_X, Y_C) = \max_{Y \in \boldsymbol{Y}}(|E_X(Y)|_\bullet) - |E_Y|_\bullet - |Y_X|_\bullet - |Y_C|_\bullet$$

the *operator increase* of `make_DAM` for E_Y, Y_X and Y_C.

As a first step, we prove that the operator bound property holds for any Divide-Structure or DAM-Structure if at most one of the substructures is too small to fulfill the operator bound property.

Lemma 4.61. Let $E_Y \in \mathcal{DAM}(Y)$, $Y_X \in \mathcal{DAM}(X)$ and $Y_C \in \mathcal{DS}$ and let $E_X =$ `make_DAM`(E_Y, Y, X) with $Y = Y_X \circ Y_C$ or $Y = Y_C \circ Y_X$. Then E_X fulfills the operator bound property (4.20) for either $|E_Y|_\iota \leq 4$ or $|Y_X|_\iota \leq 4$ or $|Y_C|_\iota \leq 2$ if the other two substructures fulfill the operator bound property.

Proof. First assume that either $E^* = E_Y$ or $E^* = Y_X$ has $n_r \leq 4$ operands. Let $\delta^\circ = \delta^\circ(E_Y, Y_X, Y_C)$ be the number of additional operators introduced by `make_DAM`. Then the number of operators of E_X is bounded by

$$|E_X|_\bullet \leq 4|E_X|_\iota - 12 - 8 + (|E^*|_\bullet - 4n_r) + \delta^\circ$$

Therefore, the operator bound property is fulfilled if

$$\delta^\circ \leq 4n_r - |E^*|_\bullet + 8 \tag{4.21}$$

We prove that this inequality is fulfilled by identifying properties of the DAM-Structure E^* depending on the number of operands and the number of operators of E^*. We guarantee these properties by an exhaustive search of all possible tree structures using Algorithm 8. For this, we compute the maximum number of operator nodes that can be created by restructuring a tree of size n_r where the resulting DAM-Structure does not fulfill the respective properties. In Table 4.23 the properties for each case and the resulting increase in operators for $E^* = E_Y$ and $E^* = Y_X$ are listed. By a simple calculation we verify that the condition in (4.21) and therefore the operator bound property is fulfilled in each of these cases.

Case	$\lvert E^*\rvert_\bullet$	Properties	δ°_Y	δ°_X
$n_r = 4$	5	$C = 0$	16	13
	≤ 4	None	20	20
$n_r = 3$	3	$C = 0$ or $D = 0$	16	13
	2	$C \in \{0, 1\}$ or $D = 0$	18	18
	1	One of A, B, C, D is in $\{0, 1\}$	18	19
	0	None	20	20
$n_r = 2$	1	$C = 0$ and (A or D is in $\{0, 1\}$) or $D = 0$ and (A or C is in $\{0, 1\}$) or $A = 0, C = 1, D = 1$	15	15
	0	$C = 0$ or $D = 0$ or ($C = 1$ and $A = 0$)	16	16
$n_r = 1$	0	$D = 1, B = C = 0$ or $A = D = 1, C = 0$ or $A = 1, B = D = 0$ or $A = D = 0, C = 1$	12	10
$n_r = 0$	0	$A = D = 1$ and $B = C = 0$	7	8

Table 4.23: The properties of a DAM-Structure E^* that have been computationally verified for the small cases ($n_r \leq 4$) of `raise` and the resulting bounds for the number of operators in a DAM-Structure created by `make_DAM`(E_Y, Y, X) for $E^* = E_Y$ or $E^* = Y_X$. A component is declared as zero or one if it does not contain operands and represents this value in the structure.

If Y_C has $n_c \leq 2$ operands, we get $|E_X|_\bullet \leq 4|E_X|_\wr - 12 - 12 + (|Y_C|_\bullet - 4n_c) + \delta^\circ$ and therefore

$$\delta^\circ \leq 4n_c - |Y_C|_\bullet + 12 \tag{4.22}$$

For $n_c = 2$ we can only achieve a complete Divide-Structure if the unique operator is a division. In this case, $|Y_C|_\bullet = 0$ and (4.22) is fulfilled. Otherwise, $F = 1$ or $G = 1$ and $\delta^\circ \leq 18$. For $n_c = 1$, there are no operations and therefore $G = 1$ and $\delta^\circ \leq 16$. Therefore, in each case the operator bound property is fulfilled and the lemma is proven. □

For `compress` the inductive bound is less tight, which makes the argument much simpler.

Lemma 4.62. Let $E_X \in \mathcal{DAM}(X)$ and $X_1, X_2 \in \mathcal{DS}$ and let $E = \texttt{make\_DS}(E_X, X)$ with $X = X_1 \circ X_2$. Then E fulfills the operator bound property for either $|E_X|_\iota \le 4$ or $|X_1|_\iota \le 2$ or $|X_2|_\iota \le 2$ if the other two substructures fulfill the operator bound property, respectively.

Proof. Let E_X have $n_r \le 4$ operands. For $n_r \ge 1$ we observe from Table 4.20 that $|E_X|_\bullet \le 4n_r - 4$. Therefore, we get

$$|E|_\bullet \le 4|E|_\iota - 8 - 8 - 4 + 10 \le 4|E|_\iota - 8$$

Now let $n_r = 0$. Then $|E_X|_\bullet = 0$ and the number of additional operators introduced by `make_DS` is ≤ 4, resulting in

$$|E|_\bullet \le 4|E|_\iota - 8 - 8 + 4 \le 4|E|_\iota - 8$$

For $X^* = X_1$ or $X^* = X_2$ with $1 \le n_c \le 2$ operands we observe that $|X^*|_\bullet \le 4n_c - 4$. Therefore,

$$|E|_\bullet \le 4|E|_\iota - 12 - 8 - 4 + 10 \le 4|E|_\iota - 8$$

and the lemma is proven. □

For $n \ge 8$ in the case of `compress` and for $n \ge 12$ in the case of `raise` we can show that at most one of the substructures violates the operator bound property. By exhaustive search we were able to show that the operator bound property holds for $3 \le n \le 8$ and $5 \le n \le 8$, respectively. Results for $n > 8$ are not achievable in a reasonable time with the hardware used. It remains to show that for `raise` the operator number after restructuring is at most $4n - 12$ for $9 \le n \le 11$. For this, we first show that we almost always save operators if we generate an incomplete DAM-Structure instead of a complete one.

Lemma 4.63. For $E_Y, \hat{E}_Y, R_1 \in \mathcal{DAM}(Y)$, $Y_X, \hat{Y}_X, R_2 \in \mathcal{DAM}(X)$ and $Y_C, \hat{Y}_C, C_1 \in \mathcal{DS}$ let E_Y and Y_X be DAM-Structures of the form $(AZ + B)/D$ with $Z \in \{X, Y\}$ and let Y_C be a Divide-Structure of the form $Y_C = F$. Let furthermore $\hat{E}_Y$, $\hat{Y}_X$ and $\hat{Y}_C$ be complete. Then

1. $\delta^\circ(E_Y, R_2, C_1) \le \max(\delta^\circ(\hat{E}_Y, R_2, C_1) - 1, 3)$
2. $\delta^\circ(R_1, Y_X, C_1) \le \max(\delta^\circ(R_1, \hat{Y}_X, C_1) - 1, 4)$
3. $\delta^\circ(R_1, R_2, Y_C) \le \max(\delta^\circ(R_1, R_2, \hat{Y}_C) - 1, 4)$

Proof. We first show various properties of any Divide- and DAM-Structures created during restructuring, which we use afterwards to show that the number of additional operators created during `make_DAM` must decrease for both E_X and E. We say that a component is zero or one if its number of operands is zero and its value in the context of the structure is zero or one. Consider a DAM-Structure and a Divide-Structure of the form

$$\frac{AX + B}{CX + D} \quad \text{and} \quad \frac{F}{G}$$

First, note that the denominator of either structure can never be zero, therefore **(1a)** $C \neq 0$ or $D \neq 0$ and **(1b)** $G \neq 0$. Since inversions are only possible in the context of a division, the numerator can never be zero or one. This yields **(2a)** $A \neq 0$ or $B \neq 0$ as well as **(2b)** $B \neq 1$ and, consequently, **(2c)** $|F|_l > 0$. Since X is retained during the operations, we have always **(3)** $A \neq 0$ or $C \neq 0$. If C is not zero, then at one point during its creation, a DAM-Structure for X was on the right hand side of a division. Since in the previous structure there was $B \neq 1$, we get **(4)** $C \neq 0 \Rightarrow D \neq 1$. Finally, creating a DAM-Structure with both $A \neq 0$ and $C \neq 0$ requires an addition in a state where $C \neq 0$, which may happen directly or during a substitution. This addition always leads to a factor in front of X and retains the value of B in the numerator, therefore **(5a)** $C \neq 0 \Rightarrow A \neq 1$ as well as **(5b)** $B \neq 0$ or $D \neq 0$.

	$Y = \hat{Y}_X + \hat{Y}_C$	$Y = \hat{Y}_X \cdot \hat{Y}_C$	$Y = \hat{Y}_X / \hat{Y}_C$	$Y = \hat{Y}_C / \hat{Y}_X$
A^*	$A(aG + cF) + B(cG)$	$A(aF) + B(cG)$	$A(aG) + B(cF)$	$A(cF) + B(aG)$
B^*	$A(bG + dF) + B(dG)$	$A(bF) + B(dG)$	$A(bG) + B(dF)$	$A(dF) + B(bG)$
C^*	$C(aG + cF) + D(cG)$	$C(aF) + D(cG)$	$C(aG) + D(cF)$	$C(cF) + D(aG)$
D^*	$C(bG + dF) + D(dG)$	$C(bF) + D(dG)$	$C(bG) + D(dF)$	$C(dF) + D(bG)$

Table 4.24: The components of the DAM-Structure $(A^*X + B^*)/(C^*X + D^*)$ created by calling `make_DAM`$(\hat{E}_Y, Y, X)$ for additions, multiplications and divisions.

In Table 4.24, the rules for the construction of a DAM-Structure by `make_DAM` depicted in Table 4.13 are revisited and extended by the rules for divisions. With the three statements from the lemma and the four possible constructions, we get a total of 12 cases that must be treated. We state the main idea for each of these cases. We denote the components of the DAM-Structure for Y as A, B, C, D and the components of the DAM-Structure for X as a, b, c, d. We say that we *save* an operation if it is present in the result for the complete structure but not for the incomplete structure. For the first statement of the lemma, we have $C = 0$. For $Y = R_2 + C_1$ and $Y = C_1/R_2$ we save at least one multiplication since with (1a) and (2c) either cF or dF contains an operand. Similarly, we save one multiplication for $Y = R_2 \cdot C_1$ with (2a) and (2c). For $Y = R_2/C_1$, we save a multiplication if $|G|_l > 0$ due to (2a). We furthermore save a multiplication if $|a|_l > 1$ or $b \neq 0$ due to (1b) and (2b). Lastly, with (1b), (2a) and (2c), we save an addition if $c \neq 0$. If neither of these statements is true, we are in Case A as listed in Table 4.25 and, hence, the operator increase is at most 3.

The proof of the other two statements is similar to first one. We briefly list the relevant steps. For the second statement, $c = 0$ and we save one multiplication for $Y = Y_X + C_1$ and $Y = C_1/Y_X$ due to (1a) and (2c). For $Y = Y_X/C_1$, we save one multiplication with (2c) and (5b). For $Y = Y_X \cdot C_1$, we either save a multiplication or an addition or with (1b), (2b), (2c), (4) and (5b) we are in Case B of Table 4.25. By that, the second statement is proven. For the third statement, we always end up in one of the small cases. For additions, we reach Case C and for multiplications (a subcase of) Case B. For divisions of the form $Y = R_1/Y_C$ we reach either Case D or Case E and for divisions of the form $Y = Y_C/R_1$ we get (a subcase of) Case A. For each operation, nearly all of the proven properties are required for the derivation. □

	E_Y	Y_1	$\circ$	Y_2	E_X	δ°
Case A	$\frac{AY+B}{D}$	F	$/$	$\frac{X}{d}$	$\frac{AX+B(dF)}{D(dF)}$	3
Case B	AY	$\frac{aX+b}{d}$	$*$	F	$\frac{A(aF)X+A(bF)}{d}$	4
Case C	Y	X	$+$	F	$X+F$	0
Case D	$\frac{Y+B}{D}$	$\frac{X}{d}$	$/$	F	$\frac{X+B(dF)}{D(dF)}$	3
Case E	$\frac{B}{Y+D}$	$\frac{X}{d}$	$/$	F	$\frac{B(dF)}{X+D(dF)}$	3

Table 4.25: The small cases for `make_DAM` occurring in the proof of Lemma 4.63 in which no operations can be saved compared to a complete structure.

Lemma 4.63 enables us to treat the cases $n = 4$ for `raise` and $n = 2$ for `compress` as if they would fulfill the operator bound property. For $n = 4$, `raise` leads to at most $4n - 11$ operators. Since the maximal case requires $C = 0$ (cf. Table 4.23), the number of operators added by `make_DAM` is reduced by at least one if at least four new operators are produced. The same applies to `compress`, which, in the case of $n = 2$, leads to at most $4n - 7$ operators with the maximal case resulting in a Divide-Structure with $G = 1$. For the inductions for `raise` in the proof of Lemma 4.59 and Lemma 4.61, an operator increase of at least 7 is assumed. Let E_X be a DAM-Structure with $n = 4$ and let E be a Divide-Structure with $n = 2$. If E_X or E is used in any of the inductive steps, then either E_X and E fulfill the operator bound property or an additional decrease in the number of operators of at least one is achieved, compensating for the additional operator in the structure. Since $\delta^\circ \geq 7$ in all of these cases, this stays true even if all structures appearing in the inductive step are replaced. Consequently, E_X and E behave equivalently to fulfilling the operator bound property. For `compress`, E_X and E can be used at will as well since neither of the bounds in Lemma 4.59 and Lemma 4.62 is tight, missing the required upper bound by 10, 2, 4 and 6 operators, respectively.

Before we prove that the operator bound property holds for `raise` with $9 \leq n \leq 11$, we show for two combinations of small structures that in these cases the operator bound property is retained even if both DAM-Structures in the inductive step do not fulfill the operator bound property.

Lemma 4.64. Let $E_Y \in \mathcal{DAM}(Y)$ with $n_r = |E_Y|_l \in \{2, 3\}$, let $Y_X \in \mathcal{DAM}(X)$ with $|Y_X|_l = 3$ and let $Y_C \in \mathcal{DS}$ fulfill the operator bound property. Then $E_X =$ `make_DAM`(E_Y, Y, X) with $Y = Y_X \circ Y_C$ or $Y = Y_C \circ Y_X$ fulfills the operator bound property.

Proof. Let $\delta^\circ = \delta^\circ(E_Y, Y_X, Y_C)$ be the number of additional operators introduced by `make_DAM`. In Table 4.26, all possible cases are listed together with computationally verified properties of the respective structures and a resulting upper bound on δ°. In each of those cases we have

$$|E_Y|_\bullet + |Y_X|_\bullet + \delta^\circ \leq 4n_r + 8$$

Case	$\lvert E_Y\rvert_\bullet$	$\lvert Y_X\rvert_\bullet$	Properties for E_Y	Properties for Y_X	δ°
$n_r \leq 3$	≤ 3	3	Not complete	$C = 0$ or $D = 0$	12
$n_r = 3$	3	≤ 2	$C = 0$ or $D = 0$	Not complete	15
	2	2	$C \in \{0, 1\}$ or $D = 0$	$C \in \{0, 1\}$ or $D = 0$	16
	≤ 1	2	Not complete	$C \in \{0, 1\}$ or $D = 0$	17
	2	≤ 1	$C \in \{0, 1\}$ or $D = 0$	Not complete	17
$n_r = 2$	1	≤ 2	$Z_1 = 0, Z_2 \in \{0, 1\}$	Not complete	13
	0	2	$Z_1 = 0$	$C \in \{0, 1\}$ or $D = 0$	14
	0	≤ 1	$Z_1 = 0$	Not complete	15

Table 4.26: The properties that have been computationally verified for DAM-Structures E_Y with two or three and Y_X with three operands, depending on the number of operators in their components. Depending on these properties, the maximum number of operators generated by `make_DAM` with E_Y, Y_X and $Y_C \in \mathcal{DS}$ is determined and listed in the last column (cf. Table 4.24).

Therefore, the number of operators in E_X is bounded as follows.

$$\begin{aligned} |E_X|_\bullet &\leq |E_Y|_\bullet + |Y_X|_\bullet + |Y_C|_\bullet + \delta^\circ \\ &\leq 4n_r + 8 + 4|Y_C|_l - 8 \\ &= 4(n_r + |Y_X|_l + |Y_C|_l) - 12 \end{aligned}$$ □

Lemma 4.65. Let T be an operator tree and let X be a subtree in T with $n = |T|_l - |X|_l$ and $9 \leq n \leq 11$. Then for $T_X = \texttt{raise}(T, X)$ it follows that $|T_X|_\bullet \leq 4n - 12$.

Proof. Let Y be the split node resulting from $\texttt{raise}(T, X)$, let $E_Y \in \mathcal{DAM}(Y)$ be the DAM-Structure for Y and let $Y_X \in \mathcal{DAM}(X)$ and $Y_C \in \mathcal{DS}$ be the children of Y created by the subroutines of `raise`. If at most one of E_Y, Y_X, Y_C does not fulfill the operator bound property, then the desired property is guaranteed by Lemma 4.59 and Lemma 4.61. Let $|Y|_l = |Y_X|_l + |Y_C|_l$. Due to the property of the split node we have $|Y|_l \geq 5$ and $|Y_X|_l, |Y_C|_l \leq 4$ for $n = 9$ and $n = 10$. There are 10 cases for $n = 9$ and 6 cases for $n = 10$ in which at least two nodes do not fulfill the operator bound property, as listed in Table 4.27.

	$n = 9$										$n = 10$					
$\lvert E_Y\rvert_l$	4	4	4	4	3	3	3	2	2	1	5	5	4	3	3	2
$\lvert Y_X\rvert_l$	4	3	2	1	4	3	2	4	3	4	4	3	4	4	3	4
$\lvert Y_C\rvert_l$	1	2	3	4	2	3	4	3	4	4	1	2	2	3	4	4

Table 4.27: Possible distributions of $n = 9$ or $n = 10$ operands during `make_DAM` such that at least two of the substructures do not fulfill the operator bound property.

By Lemma 4.63, we can treat cases in which $|E_Y|_l = 4$, $|Y_X|_l = 4$ or $|Y_C|_l = 2$ as if the respective substructure would fulfill the operator bound property. In all three remaining cases, $|Y_X|_l = 3$ and $|E_Y|_l$ is either 2 or 3. By Lemma 4.64, they fulfill the operator bound property. For $n = 11$ the property of the split node requires $|Y|_l \geq 6$ and $|Y_X|_l, |Y_C|_l \leq 5$ leading to the 8 cases shown in Table 4.28.

$n = 11$

$\|E_Y\|_l$	5	4	4	4	4	3	3	2
$\|Y_X\|_l$	4	5	4	3	2	4	3	4
$\|Y_C\|_l$	2	2	3	4	5	4	5	5

Table 4.28: Possible distributions of $n = 11$ operands during `make_DAM` such that at least two of the substructures do not fulfill the operator bound property.

As before, there is only one case that cannot be handled by Lemma 4.63, in which $|E_Y|_l = |Y_X|_l = 3$. Again with Lemma 4.64, the operator bound property is fulfilled and therefore the lemma is proven. □

Combining our previous results, we can prove an upper bound on the number of operators in a Divide-Structure created by `compress`.

Theorem 4.66. Let T be an operator tree with $|T|_l \geq 3$ and let T' be the operator dag created from T by Brent Restructuring. Then $|T'|_\circ \leq 4|T|_\circ$.

Proof. Let E be a Divide-Structure created by `compress` with $|E|_l \geq 3$ and let E_X be a DAM-Structure created by `raise` with $|E_X|_l \geq 5$. We show that E and E_X fulfill the operator bound property (4.20). For $3 \leq |E|_l \leq 8$ and $5 \leq |E_X|_l \leq 11$, this is shown by the computationally derived results in Table 4.20 and by Lemma 4.65.

Let $|E_X|_l \geq 12$ and let $E_X = \texttt{raise}(E_Y, Y, X)$. Then Y has at least six operands and its children Y_X, Y_C have at most five operands. If E_Y, Y_X, Y_C all fulfill the operator bound property, then the desired result follows from Lemma 4.59. If at most one of them does not fulfill the property, then the result follows from Lemma 4.61. Otherwise, if $|E_Y|_l \leq 3$, then $|Y|_l \geq 9$ and both $|Y_X|_l$ and $|Y_C|_l$ have at least $9 - 5 = 4$ operands. Additionally, if $|Y_X|_l \leq 3$, then $|Y_C|_l \geq 3$ and vice versa. Therefore, there is at most one DAM-Structure with less than four, or one Divide-Structure with less than two operands among E_Y, Y_X, Y_C and, with Lemma 4.63, E_X fulfills the operator bound property.

Now let $|E|_l \geq 9$ and let $E = \texttt{compress}(E_X, X)$. Then X has at least five, and its children X_1, X_2 have at most four operands. From $|E_X|_l \leq 3$ it follows that X_1, X_2 have at least $9 - 3 - 4 = 2$ operands and from $|X_1|_l = 1$ it follows $|X_2|_l \geq 4$ and vice versa. Due to the Lemmata 4.59, 4.62 and 4.63, E fulfills the operator bound property.

Since $|T|_\circ = |T|_l - 1$ and T' is created from the result of $\texttt{compress}(T)$ with at most one division, it follows that

$$|T'|_\circ \leq 4|T'|_l - 7 \leq 4(|T|_\circ + 1) - 7 \leq 4|T|_\circ$$ □

With Theorem 4.58 we have found a lower bound for the degree of invasiveness of Brent Restructuring and with Theorem 4.66 we have found a matching upper bound. In summary:

Corollary 4.67. The degree of invasiveness of Brent Restructuring is 4. □

4.3.4 Restructuring with Weighted Operands

In the previous sections, we introduced algorithms that operate locally on tree structures. In this section, we analyze the effect of a (local) restructuring algorithm on the overall expression dag. Restructuring an expression dag consists of splitting the expression dag into maximal subtrees and using the restructuring algorithms on these trees without knowledge about the underlying dag structure. This strategy produces fairly good results, especially if the number of subtrees is small compared to the total number of nodes in the expression dag. For a domain $\mathcal{D}$, we call a node in an expression dag *blocking* if it cannot be incorporated into the operator tree of its parents with respect to $\mathcal{D}$. Blocking nodes are all nodes that have been evaluated before or have more than one parent, including references that are external with respect to the current evaluation (cf. Section 4.3.1). Furthermore, depending on the restructuring algorithm, all operator nodes that cannot be used to extend a superior operator tree within the given domain are blocking nodes. For Brent Restructuring all nodes containing a root operation are blocking nodes while for (naive) AM-Balancing all nodes containing a subtraction, root or division are blocking nodes as well as all addition or multiplication nodes for which their parent does not contain the same operation. With the definition of blocking nodes, we can now specify which operator trees inside an expression dag are eligible for restructuring.

blocking node

Definition 4.68. For a domain $\mathcal{D}$, we call an operator tree in an expression dag E *maximal* in E if it does not contain blocking nodes, except for its root, and there is no set of nodes in E that can be added to the operator tree without violating this property.

maximal

Note that with this definition we allow blocking nodes that cannot be part of any operator tree in a given domain to form single-node maximal operator trees. We assume that a restructuring algorithm on a single-node operator tree is always defined and returns the node itself. The depth of an expression dag in which all maximal operator trees have been restructured depends on the number of blocking nodes as well as on the maximum indegree in the original graph.

Definition 4.69. Let E be an expression dag. The number of incoming edges for a node $v \in \mathbf{V}_0(E)$ is called the *indegree* of v.

indegree

$\mathbf{V}_l(T)$

We denote the set of operands in an operator tree T as $\mathbf{V}_l(T)$. The indegree bounds the number of edges in an expression dag and therefore relates the number of operands in an operator tree to the number of leaves of the associated expression dag.

Observation 4.70. Let E be an expression dag with n leaves and maximum indegree μ. Then the maximum number of operands in an operator tree T in E is bounded as $|\mathbf{V}_l(T)| \leq n\mu$. Moreover, since the outgoing edges of two maximal operator trees in an expression dag are always disjoint, the sum of the numbers of operands of all maximal operator trees in E is bounded by $n\mu$ as well.

If a restructuring algorithm is locally optimal, i.e., leads to a logarithmic depth of the operator tree, then the depth of a restructured expression dag can be bounded by the number of its leaves, its maximum indegree and the number of blocking nodes.

Lemma 4.71. Let R be a restructuring algorithm such that $\operatorname{depth}(R(T)) = \Theta(\log(|T|_l))$ for each operator tree T in its domain. Let E be an expression dag with n leaves, $k \geq 1$ blocking nodes and maximum indegree μ and let $R(E)$ be the result of restructuring E by applying R on each maximal operator tree in E. Then

$$\operatorname{depth}(R(E)) = \Theta\left(k \log \frac{n}{k} + k \log \mu + k\right)$$

Proof. The root of every maximal operator tree in E is either the root of E or a blocking node. Therefore, there are at most $k+1$ maximal operator trees $T_0, ..., T_k$ in E. After restructuring, for each $1 \leq j \leq k$, the tree T_j has depth $O(\log |T_k|_l)$. Since the logarithm is concave, a convex combination for numbers $a_0, ..., a_k \in \mathbb{N}$ yields

$$\sum_{i=0}^{k} \log a_i \leq (k+1) \log \left(\frac{\sum_{i=0}^{k} a_i}{k+1} \right)$$

Since furthermore the total number of operands in all operator trees is at most $n + \mu k$ and since $\log(a+b) \leq \log a + \log b + 1$ for $a, b \geq 1$, we get

$$\operatorname{depth}(R(E)) = O\left(\sum_{i=0}^{k} \log |T_i|_l \right) = O\left(k \log \left(\frac{n + \mu k}{k} \right) \right) = O\left(k \log \frac{n}{k} + k \log \mu + k \right)$$

For the lower bound, we construct an expression dag E^* by concatenating two expression dags E_1, E_2 where $\operatorname{depth}(E_1) = \Omega(k \log(n/k))$ and $\operatorname{depth}(E_2) = \Omega(k \log(\mu))$. Let T_m be an operator tree with m operands. Then there is at least one operand x_m in T_m, which has a depth of at least $\lceil \log(m) \rceil$ in $R(T_m)$. For $k_1, m_1 \geq 1$ let E_1 be the expression dag constructed from T_{m_1} by identifying x_{m_1} with the root of a copy of T_{m_1}, marking x_{m_1} as blocking and repeating this step $k_1 - 1$ times. Now for $k_2, m_2 \geq 1$ let E_2 be the expression dag constructed from k_2 copys of T_{m_2} by identifying all operands of one copy with the root of the next copy and identifying all operands of the last copy with a single leaf. In E_2, the final leaf and all root nodes, except for the first one, are blocking and each of these nodes has indegree m_2.

Finally, let E^* be the expression dag constructed by identifying the node in E_1 that has the largest depth in $R(E_1)$ with the root of E_2. Then E^* has $n = (k_1+1)(m_1-1)+1$ operands, $k = k_1 + k_2$ blocking nodes and maximum indegree $\mu = m_2$. Let $k_1 = k_2$, then the depth of E^* is given as

$$\operatorname{depth}(E^*) \geq (k_1+1)\lceil \log(m_1) \rceil + k_2 \lceil \log(m_2) \rceil + k_1 + k_2 - 1 = \Omega\left(k \log \frac{n}{k} + k \log \mu \right)$$

Note that the connections between the trees count toward the depth. Since m_1, m_2, k_1, k_2 can be chosen arbitrarily, this proves the lower bound. □

This bound can be improved by associating weights with each operand of an operator tree, reflecting the size of the subexpression at the respective operand.

Definition 4.72. Let X be an operator dag with $|X|_l = k$ and let $W \in \mathbb{R}^k$ be a k-tuple of weights greater or equal to one. Then the tuple (X, W) is called a *weighted operator dag* or, if X is a tree, a *weighted operator tree*. Let $\mathcal{D}$ be a set of operator dags. A function R, mapping each weighted operator dag (X, W) with $X \in \mathcal{D}$ to an operator dag $R(X)$ is called a *weighted restructuring algorithm* with domain $\mathcal{D}$ if for all expression dags $E_1, ..., E_k$ it follows that

weighted op. tree

weighted rest. alg.

$$\text{value}(R(X)(E_1, ..., E_k)) = \text{value}(X(E_1, ..., E_k))$$

While a weighted restructuring algorithm has more information on its operands, it still operates exclusively on an operator tree. Therefore, the notion of depth introduced in Definition 4.19 (page 55) does not adequately reflect the quality of a weighted restructuring algorithm. Consequently, we introduce a weighted depth for a weighted operator tree.

Definition 4.73. Let $(\hat{T}, W)$ be a weighted operator tree with $W = (W_1, ..., W_k)$. For the j-th operand x_j of $\hat{T}$ we call $w(x_j) = W_j$ the *weight of* x_j and for a subtree T in $\hat{T}$ we call

$w(x_j)$

$w(T)$

$$w(T) = \sum_{x \in \mathbf{V}_l(T)} w(x)$$

the *weight of* T. For a constant $t \in \mathbb{N}$ we call

$\text{depth}_\text{w}^\text{t}(T)$

$$\text{depth}_\text{w}^\text{t}(T) = \max_{x \in \mathbf{V}_l(T)} (\text{dist}_T(x) + t\lceil \log(w(x)) \rceil)$$

the *t-weighted depth* of T.

t-weighted depth

With this definition, we can equivalently represent a weighted operator tree (T, W) by (T, w). If the weight adequately measures the size of an operand, the t-weighted depth roughly represents the expected depth of the underlying expression dag if all of its maximal operator trees are restructured by the same algorithm. Consequently, a weighted restructuring algorithm should aim to create an operator dag such that the t-weighted depth of an the operator tree is minimized. With this intuition, we define what we consider an optimal restructuring strategy.

Definition 4.74. Let R be a weighted restructuring algorithm. If there is a constant t such that for each weighted operator tree (T, w) we have

$$\text{depth}_\text{w}^\text{t}(R(T)) \leq t(\lceil \log(w(T)) \rceil + 1) \tag{4.23}$$

then R is called *optimal*.

optimal

Both AM-Balancing and Brent Restructuring can be easily adjusted to produce an optimal weighted restructuring algorithm. In the case of AM-Balancing, fusing the subtrees in ascending order according to their summed-up weight is sufficient to fulfill the condition in (4.23). This can be achieved by replacing the array X in Algorithm 5 (page 76) by a priority queue and repeatedly using `make_node` on the two smallest elements in X. Algorithm 9 shows the result of these changes.

Algorithm 9: Weighted AM-Balancing.

```
Precondition: $T$ contains only nodes with operator $\circ \in \{+, \cdot\}$
Function AM_Balance($T$,$w$):
    $X \leftarrow$ priority queue
    for $x \in \mathbf{V}_l(T)$ do
        $X.insert(x, w(x))$
    end
    while $|X| \geq 2$ do
        $(x_1, w_1) = X.pop()$; $(x_2, w_2) = X.pop()$;
        $X.insert($make_node($x_1$,$x_2$,$\circ$)$, w_1 + w_2)$;
    end
    return $X.top()$;
```

Lemma 4.75. Weighted AM-Balancing is an optimal weighted restructuring algorithm.

Proof. Let (T, w) be a weighted operator tree and let T' be the operator tree resulting from restructuring. We show by induction that $\text{depth}_{\text{w}}^{1}(T') \leq \lceil \log(w(T)) \rceil + 1$. If T is a single operand, $w(T) \geq 1$ and $\text{depth}_{\text{w}}^{1}(T') = \lceil \log(w(T)) \rceil$. If T has two or more operands, then T' is of the form $X_1 \circ X_2$ with $\circ \in \{+, \cdot\}$. Due to the construction, $w(X_1) \leq w(X_2)$ and X_2 is either a single operand or an expression of the form $X_3 \circ X_4$ with $w(X_3) \leq w(X_1)$ and $w(X_4) \leq w(X_1)$. If X_2 is a single operand, then the 1-weighted depth of X_2 is at most $\lceil \log(w(T)) \rceil$. Otherwise, the weight of each of the three subexpressions X_1, X_3 and X_4 is at most half of $w(T') = w(T)$. By induction, we get

$$\text{depth}_{\text{w}}^{1}(T') \leq \max\left(\lceil \log(w(T)) \rceil + 1, \left\lceil \log\left(\frac{w(T)}{2}\right) \right\rceil + 2\right) = \lceil \log(w(T)) \rceil + 1$$

□

For Brent Restructuring, the split conditions must be adjusted to refer to the total weight of the subtree instead of the number of operands. Replacing $|Z|_l$ by $w(Z)$ for each expression Z in Algorithm 6 and Algorithm 7 (page 83) leads to the desired result.

Lemma 4.76. Weighted Brent Restructuring is an optimal weighted restructuring algorithm.

Proof. The proof is a combination of the proofs for Theorem 4.51 and Lemma 4.75. We give an overview on the main substeps of the proof. We show that for each Divide- or DAM-Structure E created during restructuring, the 5-weighted depth of the components of E is at most $5\lceil \log(w(E)) \rceil + 4$.

If there is only one operand, the 5-weighted depth of the components of E is at most $5\lceil \log(w(E)) \rceil$. For the induction step, let E_X, X_1, X_2 be the three resulting structures at the split node of `compress` or `raise` such that $w(X_1) \geq w(X_2)$. All components of E have a depth of at most 4 in the components of the substructures (cf. Table 4.12 and Table 4.13, page 84 f.). If X_1 is not a single operand, all three structures have at most half the weight of E and, by induction, the weighted depth of the resulting components is at most

$$5\left\lceil \log\left(\frac{w(E)}{2}\right)\right\rceil + 4 + 4 = 5\left\lceil \log\left(w(E)\right)\right\rceil + 3$$

Otherwise, E_X and X_2 have at most half the weight of E and the weighted depth of the resulting components is at most

$$\max\left(5\left\lceil \log\left(w(E)\right)\right\rceil, 5\left\lceil \log\left(\frac{w(E)}{2}\right)\right\rceil + 4\right) + 4 = 5\left\lceil \log\left(w(E)\right)\right\rceil + 4$$

It follows that $\mathrm{depth}_{\mathrm{w}}^{5}(E) \leq 5\lceil\log(w(E))\rceil + 5$ and therefore weighted Brent Restructuring is optimal. □

Note that both Weighted AM-Balancing and Weighted Brent Restructuring reduce to their local variants if all operands have the same weight. Therefore, we also call the local variants the *unit weight* restructuring algorithms. Now, let E be an expression dag and let T be an operator tree in E. A natural choice for the weight of an operand $x \in \mathbf{V}_l(T)$ is the number of leaves in the subexpression of E rooted at x.

unit weight

Definition 4.77. Let E be an expression dag and let E_v be the subexpression rooted at $v \in \mathbf{V}_0(E)$. Then the *leaf weight* w of v is the number of leaves in the subexpression of v, that is, $w(v) = |\mathbf{V}_0(E_v)| - |\mathbf{V}(E_v)|$.

leaf weight

For an expression dag E, let T_E be the maximal operator tree in E containing the root of E. Note that if E has n leaves and maximum indegree μ, then the leaf weight of T_E is bounded by $\mu \cdot n$ (cf. Observation 4.70). With this property, we can now show that, if the maximum indegree of the nodes in an expression dag is bounded by a constant, an optimal restructuring algorithm paired with the leaf weight leads to an asymptotic improvement compared to the unit weight (cf. Lemma 4.71).

Theorem 4.78. Let R be an optimal weighted restructuring algorithm. Let E be an expression dag with n leaves, $k \geq 0$ blocking nodes and with maximum indegree μ and let $R(E)$ be the result of restructuring E by applying R on each maximal operator tree T in R, where the operands of T are weighted with the leaf weight in E. Then

$$\mathrm{depth}(R(E)) = \Theta\left(k + \log n + k \log \mu\right)$$

Proof. Since R is optimal, there is a constant t, such that for each operator tree T

$$\mathrm{depth}_{\mathrm{w}}^{\mathrm{t}}(R(T)) \leq t(\lceil\log(w(T))\rceil + 1)$$

Let T_E denote the maximal operator tree in E containing the root of E. We show by induction that

$$\mathrm{depth}(R(E)) \leq t(\lceil\log(w(T_E))\rceil + (k+1)(\lceil\log\mu\rceil + 1))$$

If $k = 0$, then E is a tree and all operands have weight 1. Therefore,

$$\operatorname{depth}(R(E)) = \operatorname{depth}^{\mathrm{t}}_{\mathrm{w}}(R(T_E)) \leq t(\lceil \log(w(T_E)) \rceil + 1)$$

Let $k \geq 1$. For a node x let E_x denote the expression rooted at x and let k_x denote the number of blocking nodes in E_x. Since the leaf weight of T_{E_x} is bounded by $\mu \cdot w(x)$, we get by induction:

$$\begin{aligned}
\operatorname{depth}(R(E)) &= \max_{x \in \mathbf{V}_l(T_E)} (\operatorname{dist}_{R(E)}(x) + \operatorname{depth}(E_x)) \\
&\leq \max_{x \in \mathbf{V}_l(T_E)} (\operatorname{dist}_{R(E)}(x) + t(\lceil \log(w(T_{E_x})) \rceil + (k_x + 1)(\lceil \log \mu \rceil + 1))) \\
&\leq \max_{x \in \mathbf{V}_l(T_E)} (\operatorname{dist}_{R(E)}(x) + t(\lceil \log(\mu \cdot w(x)) \rceil + k(\lceil \log \mu \rceil + 1))) \\
&\leq \max_{x \in \mathbf{V}_l(T_E)} (\operatorname{dist}_{R(E)}(x) + t\lceil \log(w(x)) \rceil) + t(\lceil \log \mu \rceil + k(\lceil \log \mu \rceil + 1)) \\
&= \operatorname{depth}^{\mathrm{t}}_{\mathrm{w}}(R(T_E)) + t(\lceil \log \mu \rceil + k(\lceil \log \mu \rceil + 1))
\end{aligned}$$

Since R is optimal, we have $\operatorname{depth}^{\mathrm{t}}_{\mathrm{w}}(R(T_E)) \leq t(\lceil \log(w(T_E)) \rceil + 1)$ and therefore

$$\begin{aligned}
\operatorname{depth}(R(E)) &\leq t(\lceil \log(w(T_E)) \rceil + 1) + t(\lceil \log \mu \rceil + k(\lceil \log \mu \rceil + 1)) \\
&\leq t(\lceil \log(w(T_E)) \rceil + (k + 1)(\lceil \log \mu \rceil + 1))
\end{aligned}$$

The leaf weight of T_E can be bounded as $w(T_E) \leq \mu \cdot n$. Consequently,

$$\operatorname{depth}(R(E)) = O\left(k(\log \mu + 1) + \log n\right)$$

For the lower bound let T_m be an operator tree with m operands for each $m \in \mathbb{N}$. For $k, m_1, m_2 \geq 1$ let E^* be the expression dag created by identifying all operands of T_{m_1} with the root of a copy of T_{m_1}, repeating this step $k - 2$ times and finally identifying all operands of the last added copy of T_{m_1} with the root of T_{m_2}. Then each root node in E^*, except for the topmost one, is blocking and in each maximal operator tree all operands have the same weight. The expression dag E^* has $n = m_2$ operands, k blocking nodes, maximum indegree $\mu = m_1$, and after restructuring we get

$$\operatorname{depth}(R(E^*)) \geq k\lceil \log \mu \rceil + k - 1 + \lceil \log n \rceil = \Omega\left(k + \log n + k \log \mu\right)$$

Again, note that each of the $k - 1$ connections between the trees counts toward the depth. By that, the theorem is proven. □

With Theorem 4.78 we have shown how the depth of an expression dag can effectively be reduced if blocking nodes are present. Unfortunately, the leaf weight of a node is closely related to the operator weight introduced in Section 4.2.3 on page 70.

Lemma 4.79. Let A be an algorithm that finds the leaf weight for each node of a binary expression dag E of size n in polynomial time $T(n)$. Then the operator weight for each node of E can be found in time $O(n + T(n))$.

Proof. Let $u_1, ..., u_m$ be the operator nodes of E. We construct an expression dag E' containing operator nodes $v_1, ..., v_m$ such that the operator weight of u_i in E is equal to the leaf weight of v_i in E'. An exemplary construction is shown in Figure 4.29. First, we create nodes $x_1, ..., x_m$, that act as the leaves of E'. Then, for all $1 \leq i \leq m$ let v_i be

1. a unary node leading to x_i, if u_i does not have children in $\mathbf{V}(E)$.
2. a binary node leading to x_i and v_j, if u_i has one child $u_j \in \mathbf{V}(E)$.
3. a binary node leading to x_i and v_i', where v_i' is a binary node leading to v_j and v_k, if u_i has two children $u_j, u_k \in \mathbf{V}(E)$.

Finally, we build a binary tree over all nodes v_i. Creating E' takes linear time. The resulting graph has at most $3m$ operator nodes and m leaves. Since the leaf x_i is contained in the subexpression rooted at v_j in E' if and only if u_i is an operator node in the subexpression rooted at u_j in E, the operator weight of each node in E can be computed in time $O(n + T(4n))$. Since T is a polynomial, $T(3n) = O(T(n))$ and the lemma is proven. □

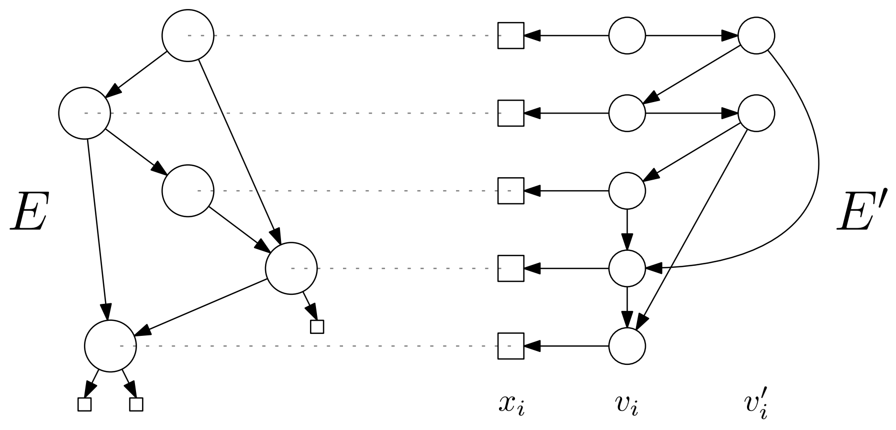


Figure 4.29: Example for the expression dag E' as created in the proof of Lemma 4.79 without the binary tree structure connecting the nodes v_i.

Since the operator weight presumably cannot be computed in subquadratic time, we can expect the computation of the leaf weight to be slow as well. As in Section 4.2.3, instead of computing the leaf weights directly, we can compute the leaf weights of the tree expansion of the expression dag.

tree leaf weight

Definition 4.80. Let E be an expression dag. For a node $v \in \mathbf{V}_0(E)$ let $w(v) = 1$ if v is a leaf and $w(v) = \sum_{(v,v') \in \mathbf{E}_0(E)} w(v')$ otherwise. Then we call w the *tree leaf weight* of v.

If clear from the context, the tree leaf weight is shortly called the tree weight. Note that the tree weight does not fulfill the bounding property of the leaf weight. In particular, it may exponentially overestimate the actual number of leaves in the expression dag. This makes it hard to store the tree weight in a primitive number type and may, in some pathological cases, lead to adverse behavior. Nevertheless, for a moderate amount of blocking nodes using the information provided by the tree weight may prove beneficial compared to purely local restructuring methods.

4.4 Experiments

In the previous sections, we introduced two conceptually different approaches to reduce the running time for the evaluation of expression-dag-based number types. Both approaches lead to asymptotic improvements. We can therefore expect large expression dags to benefit from the application of these methods to a greater extent than small expression dags. In this section, we experimentally evaluate error bound balancing and restructuring for expression dags with different structures and of different sizes and thereby compare the advantages and disadvantages of both methods.

4.4.1 Experimental Setup

All experiments in this work are performed on an Intel i7-4700MQ with 16GB RAM under Ubuntu 18.04, using g++ 7.4.0 with flags `O3` and `frounding_math`, `boost` 1.62.0 and `MPFR` 4.0.1. We use `Real_algebraic` with the configurable node type introduced in Chapter 3 as a representative example for graph-based exact-decisions number types. While there are differences between the existing number type implementations, the general behavior is similar and we expect the results from this section to be largely transferable to other number types. A comparison between `Real_algebraic` and other exact number types can be found in the dissertation of Marc Mörig [Mör15a].

The configuration of `Real_algebraic` we use as a reference is depicted in Figure 4.30. We use the interval arithmetic from `boost` as a floating-point filter and the bigfloat number type `mpfr_t` for the bigfloat approximations during the accuracy-driven evaluation. As separation bound we choose the bound by Burnikel et al. with the improvements by Pion and Yap (cf. Table 2.7 in Section 2.2.5). Experiments with the degree-measure bound variants of Li and Yap or with simultaneous use of all these bounds have not lead to any improvement on the presented running times. In fact, measure-based bounds sometimes cause underflows in the primitives used for the representation. Therefore, the mixed strategy requires special precautions to ensure correctness, which are avoided when the BFMSS[2] bound is used individually. For our experiments we always use the topological evaluation strategy to make the results more comparable and avoid uncontrollable side effects [MS15; Wil17]. For the same reason we use a cached separation bound computation strategy (cf. Section 3.2.3). Instead of the default error representation policy of `Real_algebraic`, we employ a logarithmic error representation for each stage of the evaluation. Since an error representation by a floating-point exponent is used for error bound balancing, this choice makes the results more consistent. We call the resulting configuration the *default configuration* of `Real_algebraic` and denote it by `def`. default config. Further configurations in this work are defined with respect to the default configuration by highlighting the policies that differ.

Default Configuration (def)	
LocalPolicy:	`No_local_data`
FilterPolicy:	`Boost_interval_filter_policy`
ApproximationPolicy:	`Mpfr_approximation_policy`
SeparationBound:	`Bfmss2_separation_bound`
ExpressionDagPolicy:	`Configurable_expression_dags`
RestructuringPolicy:	`No_restructuring`
EvaluationPolicy:	`Topological_evaluation`
OperationComputationPolicy:	`Default_operation_computation`
ReferenceCounterPolicy:	`Default_reference_counter`
SeparationBoundEvaluationPolicy:	`Faithful_fully_cached_evaluation`
ErrorRepresentationPolicy:	`Error_representation_by_exponent`

Figure 4.30: The default configuration for the experiments.

All data points presented in the experimental data are obtained as the average over twenty runs on different expression dags if not specified otherwise. Line charts are generated with around 50 equidistant data points where every tenth data point is marked on the line. Bar charts are generated by eight equidistant data points. Most of the shown data sets start at $x = 1000$ and end at the indicated maximum value. If the range of the results for one data point is not mentioned explicitly, it can be considered small (less than about 5% of the size of the result).

4.4.2 Fixed-Accuracy Computation

The purpose of exact-decisions number types like `Real_algebraic` is to compute a verified approximation interval for the value of an expression that is small enough to decide whether or not it is zero. Consequently, they are able to compute the value of a given expression up to an arbitrarily chosen accuracy. In contrast to an exact sign computation, a fixed-accuracy computation produces well-behaved examples, which are more suitable to observe general tendencies in an experimental context. In this section, we analyze error bound balancing and restructuring strategies for evaluations up to a fixed accuracy using partly or fully randomly generated expression dags.

Fixed-Accuracy Setup

Dividing two arbitrary floating-point numbers usually results in a number that is not representable by a single bigfloat. If rationals are used as operands for the expression dag, side effects caused by bigfloat conversions of exact intermediate results can largely be avoided. For this reason, the experimental results presented in this section are exclusively based on expression dags over random non-zero rationals. Each of these operands is represented as an expression dag itself, consisting of one division node and two floating-point numbers. To prevent the operands from being changed during restructuring, each division node is provided with an additional parent node, which is external with respect to the evaluated part of the expression dag. The underlying floating-point numbers are generated using an exponential distribution around $\lambda = 1$. With this distribution, the expected value after applying a series of multiplications and divisions is 1 and therefore we can expect the magnitude of the result to stay roughly the same.

The operators used during the experiments are usually chosen randomly and with equal probability from additions, multiplications and divisions. We do not use subtractions, negations or root operations. Subtractions behave similar to additions in each aspect that is relevant to our scenarios and negations do not have any relevant effect on the performance or on the used methods at all. Both operations, however, may cause zeros and therefore errors caused by a division by zero. Root operations cause the separation bound to shrink very quickly and to potentially underflow. While we do not explicitly use exact computation in this section, the denominators of divisions must be guaranteed to be unequal to zero. This mechanism could be deactivated for our purposes. However, incorporating root operations does not add much explanatory power to the experiments. For error bound balancing methods, the influence of a node containing a root operation is comparable to the influence of other operator nodes, while for restructuring methods the function as blocking node can be simulated by adding an additional reference to a node of any other type.

Error Bound Balancing

Consider a list-like expression dag E_{list} with randomly chosen additions, multiplications and divisions (cf. Figure 4.4, page 56). Error bound balancing significantly reduces the absolute cost induced by its unfavorable structure. We compare the standard error distribution with the path weight error distribution (`pwebb`) and the error distribution resulting from the generic error distribution in (4.15) using the tree weight (`twebb`). Figure 4.31 shows the configurations associated with these two error bound balancing strategies by highlighting their respective differences to the default configuration.

pwebb : def	**twebb : def**
ErrorDistributionPolicy: `Path_weight_distribution` **ErrorRepresentationPolicy**: `Balanced_error_representation`	**ErrorDistributionPolicy**: `Tree_weight_distribution` **ErrorRepresentationPolicy**: `Balanced_error_representation`

Figure 4.31: Configurations for number types with error bound balancing, either with the path weight error distribution (`pwebb`) or with the error distribution based on the tree weight (`twebb`).

Figure 4.32 shows the average time needed for the evaluation of a list-like expression dag to accuracy $z = -1000$ depending on the number of operator nodes. The evaluation time with the standard error distribution shows a distinct quadratic growth, while both error bound balancing strategies display an almost linear behavior. The tree weight method is about 1–3 % faster than the path weight method, presumably due to a reduced overhead. However, compared to the total cost reduction, this difference is barely noticeable. For

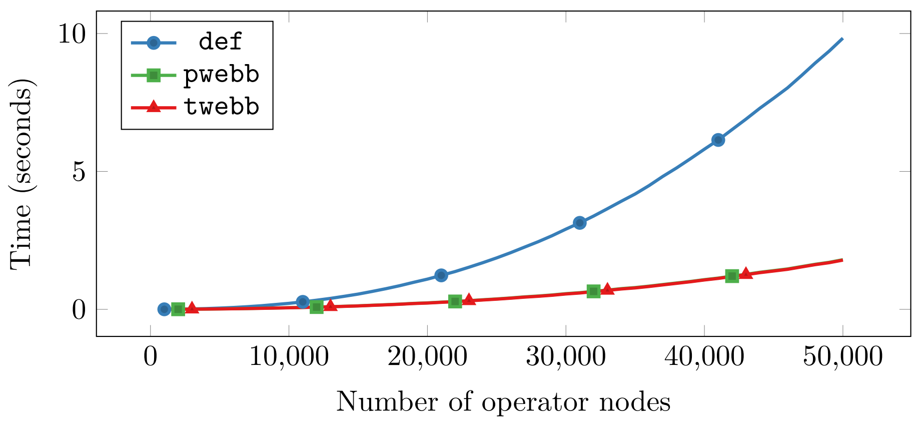


Figure 4.32: Running times of error bound balancing methods for the evaluation of a list-like expression dag with random operators to a target accuracy of $z = -1000$. Both error bound balancing methods lead to a significant decrease in evaluation time.

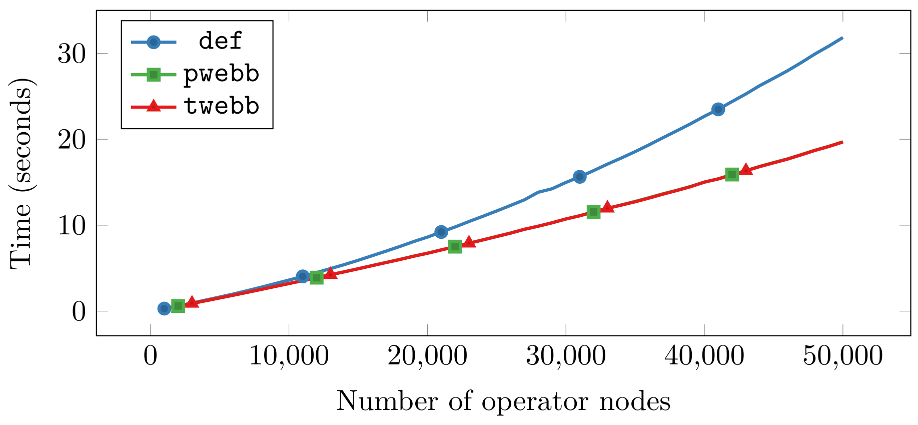


Figure 4.33: Running times of error bound balancing strategies for the evaluation of a random list-like expression dag with random operators and target accuracy $z = -100000$. Error bound balancing achieves a considerable reduction in evaluation time, although relatively less than for a lower target accuracy. No significant differences between the two error bound balancing strategies are present in the data.

small numbers of operator nodes (up to around $n = 1000$), the evaluation times for all distributions are similar. The cost induced by the target accuracy at the root node is not affected by the choice of the error distribution (cf. Corollary 4.34, page 66). Therefore, the relative performance gain due to error bound balancing diminishes if the target accuracy is high compared to the number of operator nodes. Figure 4.33 shows the evaluation times of the same experiment as before with a 100 times higher target accuracy. While the general behavior is still similar, the relative gain is reduced from over 80 % to around 40 % for 50000 operator nodes. If the list-like expression dag is replaced by a balanced expression dag, the standard error distribution already leads to near-optimal

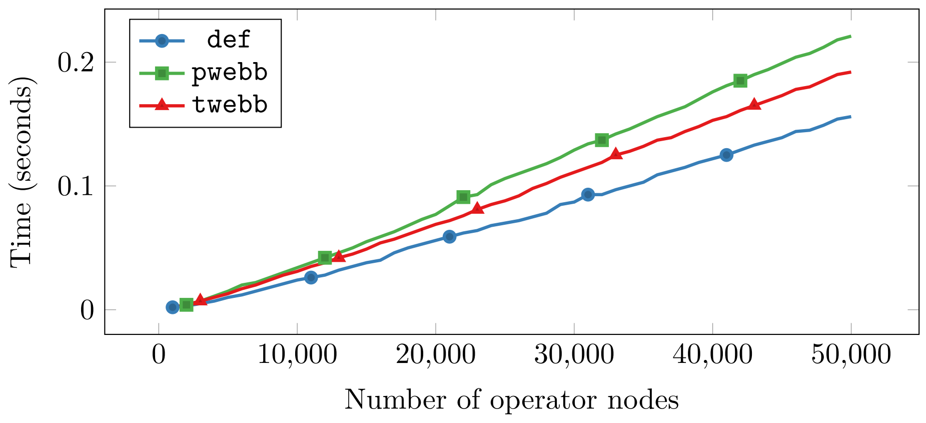


Figure 4.34: Running times of error bound balancing strategies for the evaluation of a balanced expression tree with random operators with target accuracy $z = -1000$. Both error bound balancing method increase the running time due to the induced overhead.

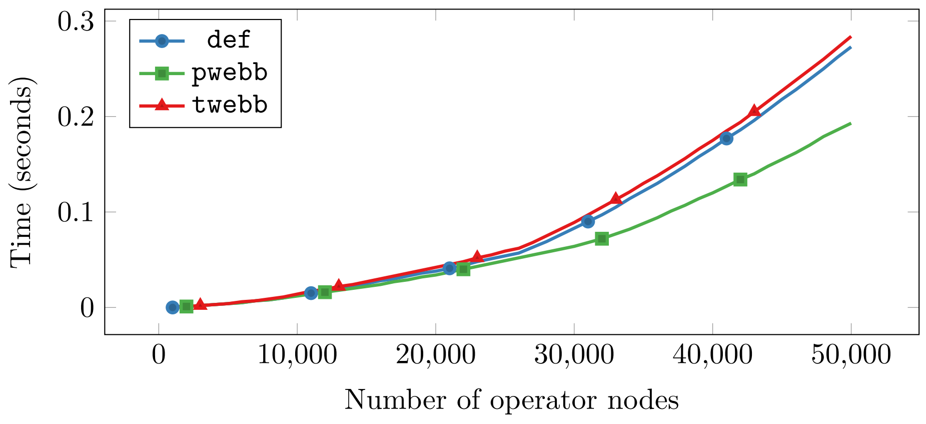


Figure 4.35: Running times for evaluating E_{lps} to accuracy $z = -1000$ with different error bound balancing strategies. The error distribution based on the tree weight behaves identically to the standard error distribution. The path weight error bound balancing reduces the running time by up to 30 % for large numbers of operator nodes.

cost. In this case, the computation of a balanced error distribution does not have much impact, but adds an overhead to the computation. If the target accuracy is small, the overhead becomes significant relative to the total cost of the evaluation. In Figure 4.34, the evaluation times of balanced expression dags for a target accuracy of $z = -1000$ are shown. The standard error distribution is preferable in this case. Computing the tree weight adds an overhead of around 30 % to the computation. The computation of the path weight error distribution is more expensive than the heuristic approach, producing an overhead of up to 40 % for the displayed values.

The main difference between the path weight and the tree weight heuristic becomes evident if the expression dag contains common subexpressions. Let E_{lps} be an expression dag consisting of one single operand and n additions, such that both summands of the k-th addition refer to the node containing the $(k-1)$-th addition (cf. Figure 4.10, page 74). When evaluating E_{lps}, we observe that the error distribution based on the tree weight heuristic leads to similar running times as the standard error distribution, whereas the path weight distribution can significantly reduce the running time (Figure 4.35). The tree weight virtually expands the expression dag into a tree before counting the operator nodes. Thereby, it exponentially overestimates the impact on the total evaluation cost of nodes the closer they are to the root. Randomly created expression dags tend to be almost balanced. Experiments on random expression trees without common subexpressions therefore reproduce the results from Figure 4.34. If common subexpressions are introduced into the randomly generated graphs, the balanced error distributions start to slightly outperform the standard error distribution. With similar conditions as in the previous experiments, error bound balancing reduces the running time by about 5 %. Surprisingly, due to the balanced nature of randomly generated dags, even for a very high rate of common subexpressions the tree weight leads to similar results as the path weight.

Local Restructuring

We compare the default configuration of `Real_algebraic` with configurations supporting AM-Balancing and Brent Restructuring with unit weights as shown in Figure 4.36. Similar to error bound balancing, the impact of restructuring is higher if the size of

amb : def
BalancingCondition: `Balance_addition_and_multiplication` **RestructuringPolicy**: `Balance_same_operation`

uwbre : def
BrentSplitCondition: `Split_unit_weight` **RestructuringPolicy**: `Brent_restructuring`

Figure 4.36: Configurations for the default number type with additional restructuring methods, implementing either AM-Balancing (`amb`) or (unit weight) Brent Restructuring (`uwbre`).

the expression dag increases. Furthermore, the cost induced by the target accuracy is usually not positively affected by restructuring and therefore the improvement might be overshadowed if a high overall accuracy is required. In contrast to error bound balancing, each node in an expression dag can at most once be part of a restructuring process. Repeated accuracy requests on the same expression dag, as performed during the cascaded filtering process in exact-decisions number types, reduce the relative impact of the overhead caused by restructuring.

AM-Balancing can only be useful if there are large subtrees in the expression dag that consist exclusively of additions or exclusively of multiplications. If large parts of the expression dag have this property, we can expect it to produce a near-optimal result. Figure 4.37 shows the evaluation time of list-like expression dags consisting exclusively of multiplications. Both AM-Balancing and Brent Restructuring significantly improve on the evaluation time of the default configuration. Although not visible in the data,

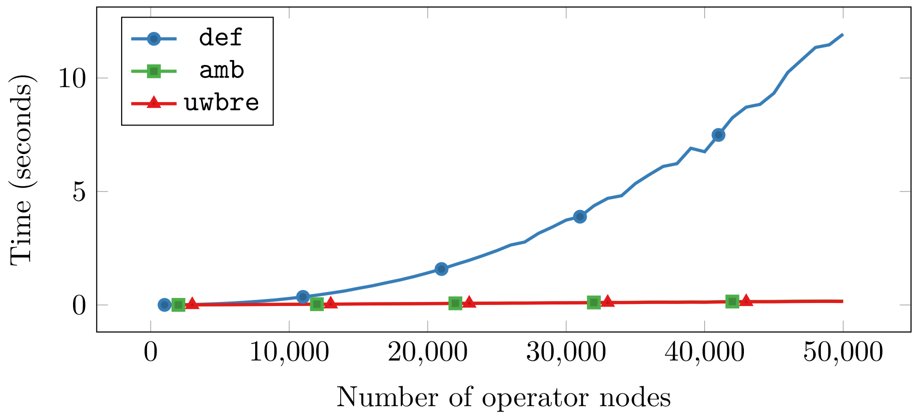


Figure 4.37: Running times with AM-Balancing and Brent Restructuring for the evaluation to accuracy $z = -1000$ of a list-like expression dag consisting exclusively of multiplications. Both restructuring strategies reduce the running times by up to 98 %.

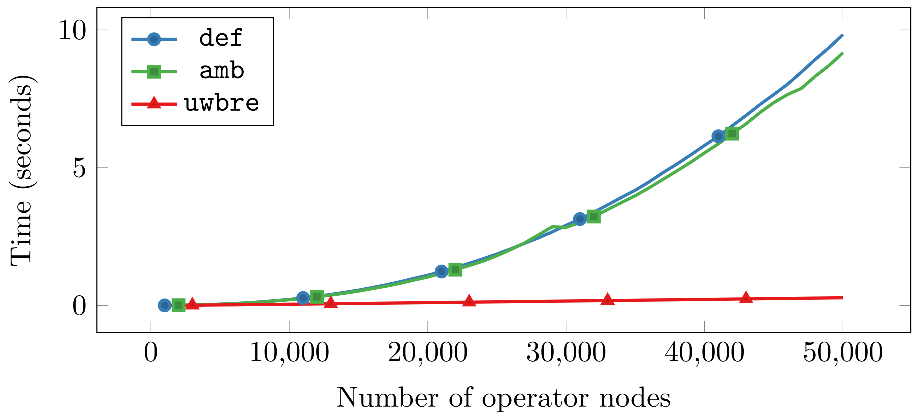


Figure 4.38: Running times with AM-Balancing and Brent Restructuring for the evaluation to accuracy $z = -1000$ of a list-like expression dag with random operators. Brent Restructuring strongly reduces the running time for large numbers of operator nodes. AM-Balancing has a small positive effect for large numbers of operator nodes, resulting in a maximum improvement of about 7 %. For 1000 operator nodes, Brent Restructuring does not improve, and AM-Balancing slightly worsens the running time due to the additional overhead.

the restructuring time for AM-Balancing is significantly lower (by about 50 %) than the restructuring time for Brent Restructuring. As soon as long sequences of nodes with the same operator are interrupted by nodes with other operators, AM-Balancing is rendered nearly useless. When the operators are chosen randomly from additions, multiplications and divisions, only Brent Restructuring still leads to a significant improvement as shown in Figure 4.38. Due to the more complicated structure, Brent Restructuring adds several additional operator nodes to the expression dag. The total number of operator nodes after restructuring is usually more than twice as high as in the original graph. Compared to the previous scenario, the running time is almost doubled, reducing the gain for a large

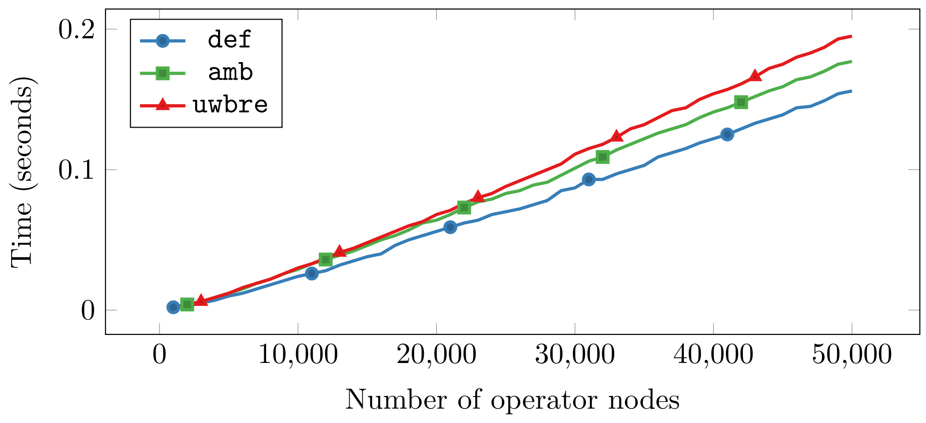


Figure 4.39: Running times with AM-Balancing and Brent Restructuring for the evaluation to accuracy $z = -1000$ of a balanced expression dag with random operators. Both restructuring strategies increase the running time. For large numbers of operator nodes, the running time is increased by about 13 % for AM-Balancing and 25 % for Brent Restructuring.

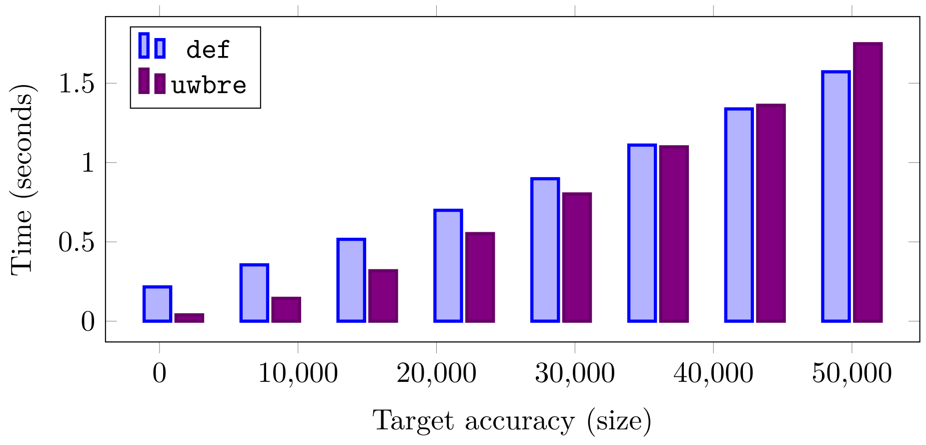


Figure 4.40: Running times with Brent Restructuring for the evaluation of a list-like expression dag with 10000 operator nodes, random operators and different target accuracies. Brent Restructuring performs worse if the minimum cost per node is increased. While for low target accuracy Brent Restructuring reduces the running time by over 80 %, the running time is increased by 10 % for $z = -50000$.

number of operator nodes from over 98 % to about 97 %. At the same time, for small numbers of operands the running time is slightly worsened by Brent Restructuring.

If the expression dag is already balanced, neither of the restructuring methods improves on the evaluation time. Figure 4.39 shows the evaluation times for balanced expression dags with uniformly distributed additions, multiplications and divisions. By balancing subtrees of additions or multiplications while ignoring the size of their operands, AM-Balancing can slightly worsen the depth of the overall expression. Brent Restructuring leads to a significant increase in evaluation time by creating DAM-Structures and thereby increasing the number of operator nodes by about 20 %. Unnecessary restructuring of already balanced parts of the expression dag can be easily avoided with the introduction of a restructuring threshold, which prevents subtrees to be restructured if the depth of the expression dag is already almost logarithmic in its size. Unfortunately, due to the increase in the number of operator nodes, Brent Restructuring can lead to a significantly worse result than the default configuration even if restructuring is useful in general. A larger initial requested accuracy results in an increase in the fixed cost of the evaluation of an operator node. If the target accuracy for the evaluation is high compared to the number of operator nodes, applying Brent Restructuring can be detrimental to the running time. In Figure 4.40, the evaluation time of a list-like expression dag with 10000 random operator nodes is shown for different target accuracies. The increase in evaluation time is roughly linear in both cases with the slope for restructuring being about 1.25 times the slope for the default configuration. Note that this difference is remarkably low, considering that Brent Restructuring leads to twice the number of operator nodes. During restructuring, expensive divisions are replaced by less expensive multiplications, reducing the evaluation cost of these operations (cf. Section 2.1.2).

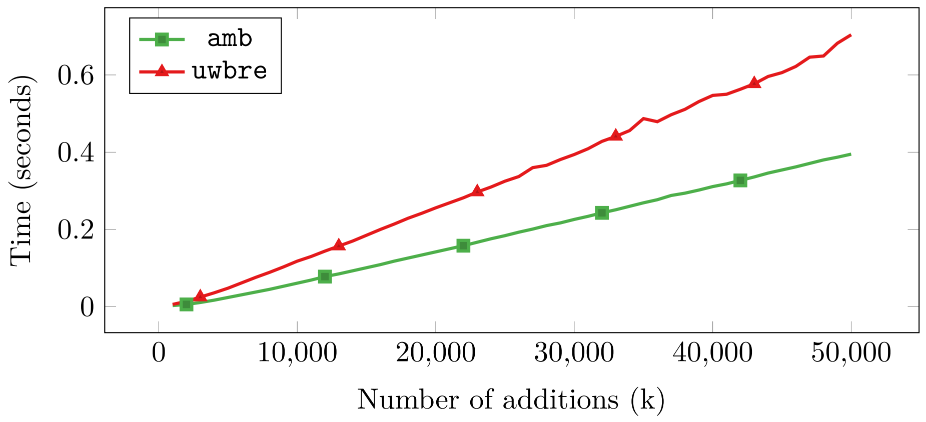


Figure 4.41: Running times for the evaluation of E_{div}^{k} to accuracy $z = -1000$ with AM-Balancing and Brent Restructuring. AM-Balancing is almost twice as fast as Brent Restructuring since Brent Restructuring incorporates the division nodes into the restructuring process and thereby creates a large amount of additional operator nodes.

In Lemma 4.52 in Section 4.3.3 (page 86), it was made evident that in some cases Brent Restructuring may lead to significantly worse results than AM-Balancing, regardless of the target accuracy. Figure 4.41 shows the results for the evaluation of an expression dag E_{div}^{k}, which consists of a linear list of additions over k divisions of two operands. In consistence with Lemma 4.52, Brent Restructuring creates 2.5 as many operator nodes as AM-Balancing. The relative difference in the evaluation time between both methods grows as the number of operands gets bigger. For a small requested accuracy and a small number of operands, the additional evaluation cost caused by the additional operator nodes is overshadowed by the maintenance cost of the data structure. With either of the two factors increasing, the evaluation time with Brent Restructuring eventually settles at twice the time of the evaluation with AM-Balancing.

Weighted Restructuring

Weighted restructuring methods as introduced in Section 4.3.4 can in theory produce a better structure than restructuring methods with only local information. We compare Brent Restructuring with unit weights with Weighted Brent Restructuring methods using the leaf weight and the tree (leaf) weight (Figure 4.42). The consequences of switching to a weighted adaptation for AM-Balancing can be expected to be similar to the respective consequences for Brent Restructuring.

lwbre : def
BrentSplitCondition: `Split_leaf_weight` **RestructuringPolicy**: `Brent_restructuring`

twbre : def
BrentSplitCondition: `Split_tree_weight` **RestructuringPolicy**: `Brent_restructuring`

Figure 4.42: Configurations for the default number type with weighted restructuring methods, implementing Weighted Brent Restructuring with the leaf weight (`lwbre`) and the tree (operand) weight (`twbre`).

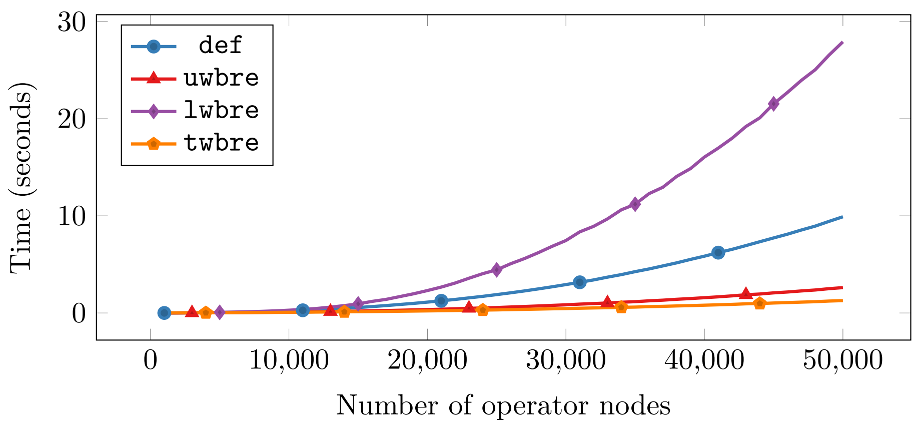


Figure 4.43: Running times for different Weighted Brent Restructuring implementations for the evaluation of a list-like expression dag with random operators and 5 % blocking nodes to target accuracy $z = -1000$. The computation of the leaf weight strongly increases the running time. Both the unit weight approach and the tree weight approach reduce the running time. For large numbers of operator nodes, the configuration based on the tree weight needs less than half the time of the configuration based on the unit weight.

Blocking nodes in an expression dag can be simulated by adding an additional external reference to the respective node just as we do for the division nodes in the operands. Figure 4.43 shows the results of an evaluation of list-like expression dags with uniformly distributed operations and with 5 % of the operator nodes randomly marked as blocking nodes. Computing the leaf weight in a naive way by a depth-first traversion of each subexpression at a blocking node turns out to be far from a feasible solution. Even for a small number of blocking nodes, the time needed for the weight computation exceeds the gain from switching to a weighted method. For large numbers of operands, even the gain from restructuring in general is negated. The tree weight heuristic on the other hand can improve on the unit weight method. Without common subexpressions, the tree weight is identical to the leaf weight and its size is bounded by the size of the expression dag. Therefore approximate computations are not necessary. With increasing numbers of operands, the weighted restructuring method based on the tree weight reduces the running time by more than 50 % compared to the unit weight approach. Nevertheless, the running time for `twbre` is still 20 times as high as the running time for `uwbre` in the case where no blocking nodes are present (cf. Figure 4.38).

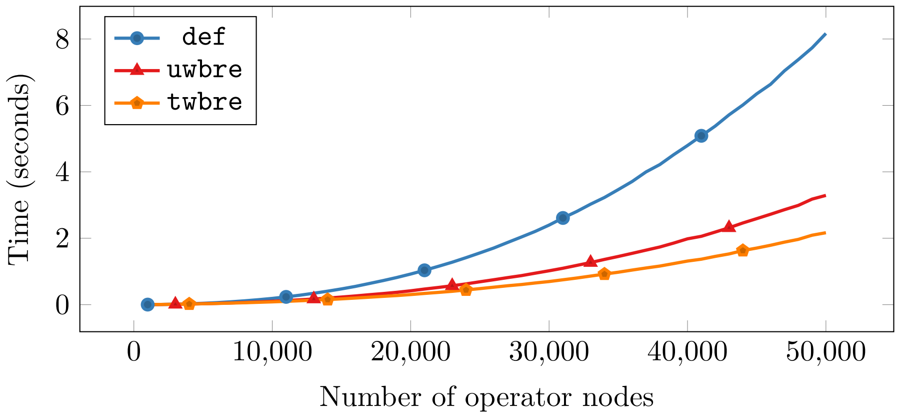


Figure 4.44: Running times for Weighted Brent Restructuring implementations for the evaluation of a list-like expression dag with random operators and 10 % loops to target accuracy $z = -1000$. Both the unit weight and the tree weight approach reduce the running time considerably. The configuration with the tree weight is up to 35 % faster than the configuration with the unit weight.

For a small number of common subexpressions, the tree weight heuristic still performs reasonably well. In Figure 4.44, evaluation times are shown for a list-like graph where 10 % of the nodes are loops, i.e., have both children point to the same subexpression. The standard operators are distributed uniformly along the graph with the exception of the loop nodes, which are exclusively additions to prevent the expression from degenerating as well as to avoid an exponential increase or decrease in the size of the approximations. Using the tree weight is beneficial in this case compared to the unit weight. While overall far too slow and therefore left out from the figure, the leaf weight leads to a more balanced structure than the tree weight and improves the bigfloat evaluation time by an additional 5–10 %.

Internal Versus External Balancing

We briefly compare error bound balancing to restructuring methods in their effectiveness of dealing with bad structure. Error bound balancing can be understood as "internal" with respect to the evaluation process, whereas restructuring methods are "external" in this regard. In consequence, error bound balancing can improve even on very complicated or already evaluated structures. Restructuring, on the other hand, is able to improve the structure on a more fundamental level and therefore may lead to larger performance gains. In a list-like expression dag with random operators, both methods significantly improve on the default configuration (cf. Figure 4.32 and Figure 4.38). In Figure 4.46, a comparison is depicted between the evaluation times for a configuration using the path weight error distribution and a configuration using Brent Restructuring with unit weights. While error bound balancing leads to an optimal distribution of the variable cost, restructuring balances the cost increase due to the operator constants as well and therefore leads to a better evaluation time. Using a combined configuration `cmb` that utilizes both error bound balancing and restructuring as shown in Figure 4.45 does not improve further on the evaluation time obtained by restructuring. Since restructuring already leads to a largely balanced expression dag, error bound balancing is mostly useless, but adds additional overhead (cf. Figure 4.34). If, however, blocking nodes are present, the relation between error bound balancing and restructuring is reversed. Figure 4.47 shows the results for the evaluation of a list-like expression dag with 10 % loops as previously used for the comparison of restructuring methods (cf. Figure 4.44). Error bound balancing is not affected by the additional

cmb : def

ErrorDistributionPolicy:
`Path_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`
BrentSplitCondition:
`Split_unit_weight`
RestructuringPolicy:
`Brent_restructuring`

Figure 4.45: Configuration for a number type implementing both path weight error bound balancing and unit weight Brent Restructuring.

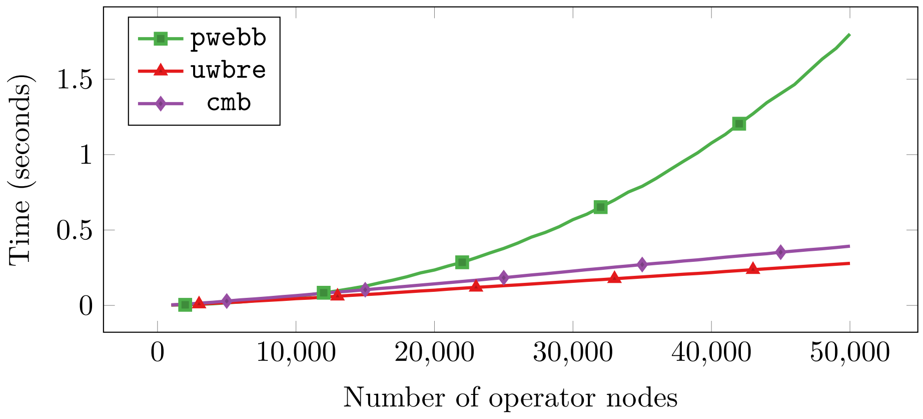


Figure 4.46: A comparison between the running times of error bound balancing and Brent Restructuring on list-like expression dags with random operators and target accuracy $z = -1000$. Brent Restructuring is up to 85 % faster than error bound balancing due to the fixed cost increase. Combining both strategies does not improve on the running time of Brent Restructuring, but performs worse. For large numbers of operands, the combined strategy is about 30 % slower than pure restructuring.

blocking nodes. Restructuring methods, on the other hand, are less effective in creating a balanced graph structure. Combining both methods leads to a significant improvement compared to the individual application of either strategy. While restructuring reduces the impact of operator constants in the maximal subtrees, error bound balancing reduces the overall variable cost to a minimum. Through the use of weighted restructuring, the evaluation time can be further reduced.

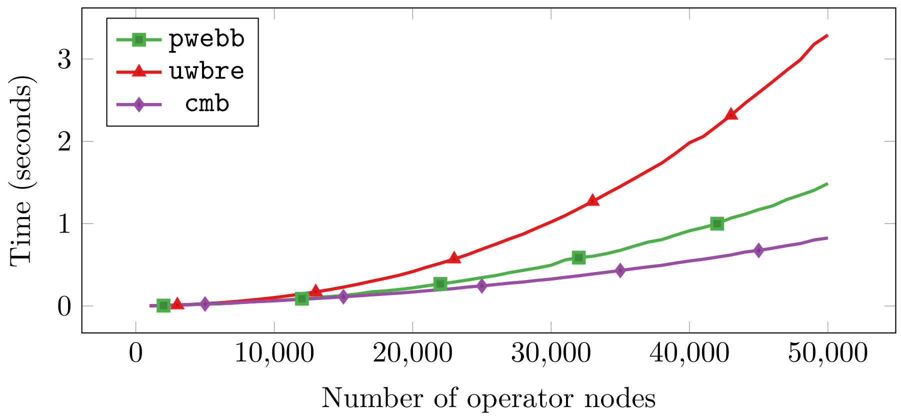


Figure 4.47: A comparison between the running times of error bound balancing and Brent Restructuring on list-like expression dags with random operators, 10 % loops and target accuracy $z = -1000$. The configuration with error bound balancing takes less than half the time of the configuration with restructuring for each data point. Combining both strategies leads to an additional reduction in the running time. The combined configuration is around 40 % faster than `pwebb` and more than 70 % faster than `uwbre`.

4.4.3 Exact Computation

While fixed accuracy scenarios are well-suited for experiments, the actual purpose of exact-decisions number types lies, as the name suggests, in making exact decisions. The running time in this case depends not only on the running time of a single evaluation but also on the maximum accuracy needed to make the decision and the behavior of the number type over a sequence of evaluations with progressively increasing target accuracies. The process of making decisions always reduces to computing the sign of an expression (cf. Section 2.2.1). The expression is then evaluated with increasing target accuracy until the value can either be separated from zero or a separation bound is hit (cf. Section 2.2.6). The impact of balancing methods is expected to be very different depending on whether or not the actual value of the expression is equal to zero. If the value is not zero, generally, a small number of iterations is sufficient to make a decision. In this case the impact of bad structure in the expression is high compared to the impact of the target accuracy. Otherwise, the target accuracy grows until it eventually hits the separation bound, which in turn shrinks considerably with the number of nodes. With increasing target accuracy, the impact of structure and therefore the impact of balancing methods diminishes. In this section, we exclusively perform experiments where the sign of the evaluated expression dag is zero. We create expression dags in the same way as for the fixed-accuracy experiments, except for the choice of the operands. Instead of rational numbers, we use simple exponentially distributed primitive floating-point numbers in order to create a more realistic setting. From an arbitrary expression dag E we obtain an expression dag E_0 with sign zero by creating an exact copy of E and comparing both expression dags for equality. In the following, descriptions are always given with respect to the expression dag E, whereas the experiments are performed on E_0.

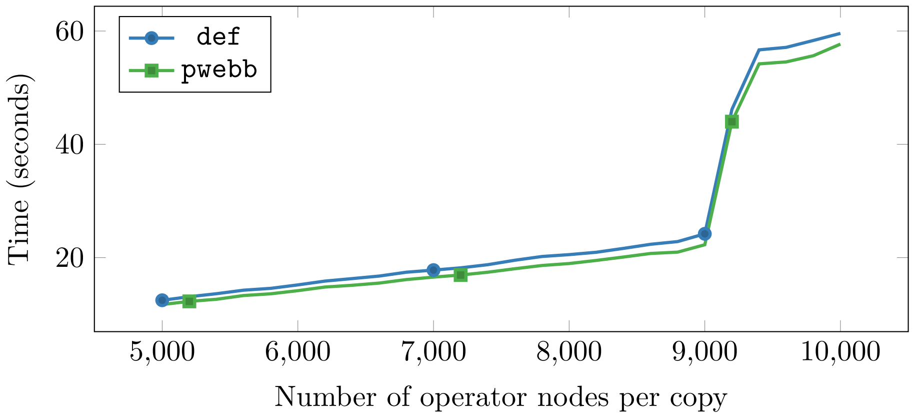


Figure 4.48: Running times with and without error bound balancing for the comparison of two identical list-like expression dags with random operators. Error bound balancing slightly reduces the running time. At around 9000 nodes per copy, a jump in running time happens where an additional iteration of the floating-point filter is executed before the error interval is sufficiently small to fall below the separation bound.

While seldom disadvantageous, error bound balancing has little effect on the overall running time of a decision-making process if the resulting sign is zero. The running time for a list-like expression dag with random operators is improved by about 5 % on average as shown in Figure 4.48. For already balanced expression dags, a small increase in running time by about 1 % can be noted. For 10000 nodes, the accuracy increase along the edges only accounts for about 0.2 seconds of the running time of a single evaluation (cf. Figure 4.32). Error bound balancing reliably reduces this cost, which sums up to about two seconds. However, since most of the running time is caused by final target accuracy at the separation bound, the relative improvement is still mediocre. On the contrary, restructuring leads to a surprisingly strong improvement for list-like expression dags as shown in Figure 4.49. While a computation with the default strategy quickly becomes infeasible when the size of the expression dag grows, the running time after applying Brent Restructuring shows only a very mild increase. The reason for this effect lies in an improved order of the operator nodes. Brent Restructuring replaces all divisions by multiplications, save for one final division in the root node. Aside from being more expensive than multiplications, divisions frequently lead to values that cannot be represented as a floating-point number. In the restructured graph, all subexpressions are eventually representable by bigfloats. As soon as the required precision is reached, bigfloat conversions take place, which significantly reduce the complexity of the expression dag.

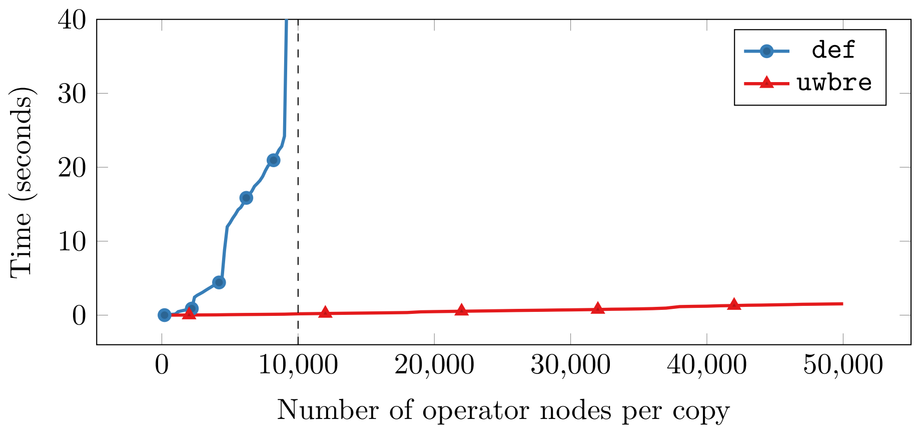


Figure 4.49: Running times with and without Brent Restructuring for the comparison of two identical list-like expression dags with random operators. Restructuring drastically reduces the running time. For 10000 operands per copy, `uwbre` needs less than 0.5 % of the running time of the default strategy. The running time of Brent Restructuring increases from on average 0.008 s for 1000 operands to 1.552 s for 50000 operands.

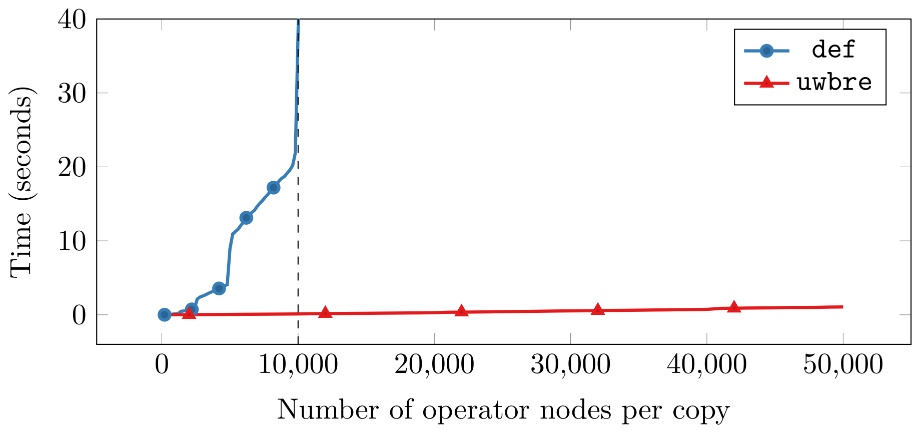


Figure 4.50: Running times with and without Brent Restructuring for the comparison of two identical random expression dags with random operators. As for the list-like expression dag, a drastic reduction in running time through restructuring is shown. The running times are around 30–40 % lower than in the unbalanced case for both strategies.

Note that this process does not require a separation bound since bigfloat types can decide whether an executed operation was exact. Creating a balanced structure additionally reduces the size of the floating-point representations of most subexpressions and therefore accelerates this process. Still, the same effect can be observed for already balanced expression trees. Figure 4.50 shows the running times for random expression dags, which usually are mostly balanced. The running time for both the default configuration and the configuration with restructuring is up to one third smaller than for a list-like expression tree, leading to the same general behavior. Note that in all cases the final separation bound and therefore the maximum target accuracy is similar to the separation bound for the default configuration. For a list-like expression dag with 10000 operator nodes, the size of the magnitude of a typical separation bound is only decreased by about 1 %. Therefore, the improvement is not primarily caused by an improved separation bound.

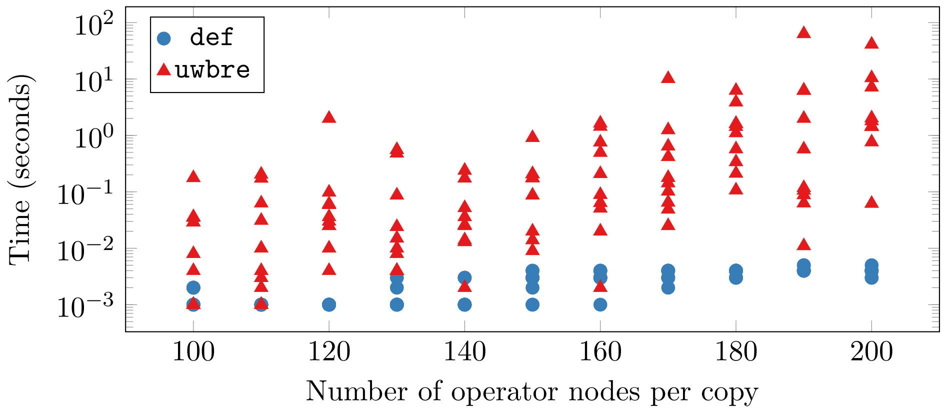


Figure 4.51: Running times with and without Brent Restructuring for the comparison of two identical list-like expression dags with random operators and 5 % nodes with an additional external reference. Each marker represents a single data point. The results for `uwbre` vary widely (note the logarithmic scale) and almost always increase the running time considerably.

The positive effect of restructuring can be dramatically reversed as soon as blocking nodes are present. Figure 4.51 shows running times for a set of list-like expression trees with 5 % blocking nodes each. The running times after restructuring differ widely, from times comparable to the running time of the default configuration up to running times which are five decimal orders of magnitude higher. Brent Restructuring introduces common subexpressions to the expression dag. While in general a multiple usage of a subexpression does not increase the running time of a single evaluation, it can cause the separation bound to shrink very fast if the subexpression contains both additions and divisions. The adverse side effects of restructuring in the presence of blocking nodes can be mitigated by a weighted restructuring method. By isolating the blocking nodes, the number of nodes affected by them decreases and, as in the balanced case, more bigfloat conversions happen. Nevertheless, the separation bound may still shrink considerably, although it is generally larger than for standard Brent Restructuring. Results for Weighted Brent Restructuring on a list-like expression dag with 5 % blocking nodes are shown in Figure 4.52. Note that the maximum size of the trees in the presented data points is higher than in Figure 4.51. In contrast to unit weight restructuring, exact decisions for expression dags with up to 200 nodes can still be computed in a reasonable time (< 0.5 s). For a slightly higher number of nodes, however, the evaluation becomes infeasible as well.

If the expression dag is already balanced and contains blocking nodes, restructuring again reliably improves the running time as shown in Figure 4.53. While still decreased when compared to the standard configuration, the separation bound turns out to be much larger overall. At the same time, the structure of the expression dag allows for more bigfloat conversions since a single blocking node prevents at most a logarithmic number of other nodes from having an exact floating-point representation. These effects lead to the somewhat paradoxical situation that restructuring is more useful the more balanced the expression dag already is, especially if blocking nodes are present.

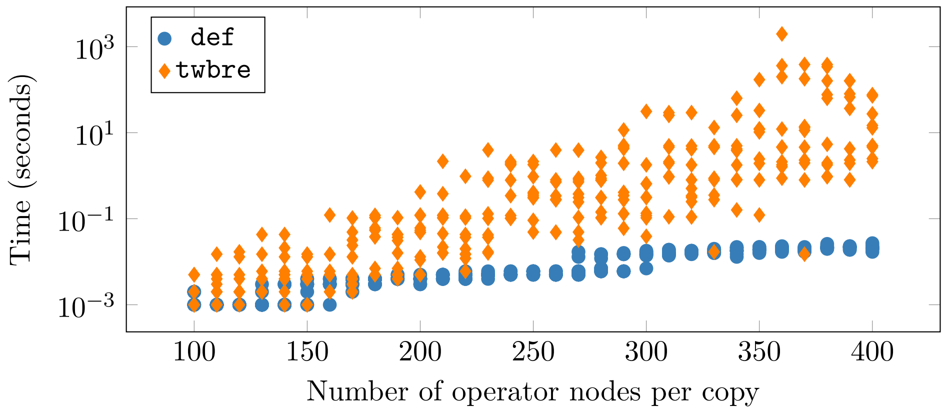


Figure 4.52: Running times with and without Weighted Brent Restructuring for the comparison of two identical list-like expression dags with random operators and 5 % nodes with an additional external reference. Each marker represents a single data point. The results for `twbre` vary widely and almost always increase the running time. It takes about twice as many operands as for `uwbre` to arrive at a similar running time distribution.

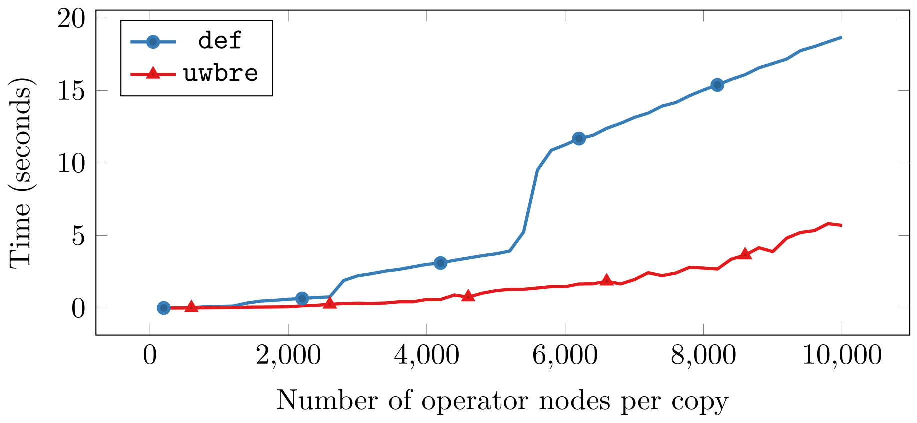


Figure 4.53: Running times with and without Brent Restructuring for the comparison of two identical balanced expression dags with random operators and 5 % nodes with an additional external reference. Restructuring reduces the running time by 70–87 %.

At the end of this section, we have a look at two examples, which are slightly more realistic in the sense that the structure, the occurring values and the distribution of the operations are closer to what one might expect in a real application. In the first example we compute the n-th Fibonacci number F_n in two different ways, once by its recursive definition and once by its closed form

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}, \text{ where } \varphi = 1 - \psi = \frac{1+\sqrt{5}}{2}$$

Afterwards, we compare both results. In the second example, we verify the binomial theorem

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

for $x = \sqrt{13}$ and $y = \sqrt{17}$. In both cases, several design choices can be made by the programmer, which influence both the general efficiency and the impact of balancing strategies. We use the most straightforward approach for both cases and disregard more elaborate approaches such as exponentiation by repeated squaring. The source code for both tests is displayed in Figure 4.54.

```
template <class NT>
void fibonacci(const int n){
  NT sqrt5 = sqrt(NT(5));
  NT phi = (NT(1) + sqrt5) / NT(2);
  NT psi = (NT(1) - sqrt5) / NT(2);

  NT phiN = phi; NT psiN = psi;
  NT fib0 = 0; NT fib1 = 1; NT tmp;
  for(int i = 1; i < n; ++i){
      tmp = fib1;
      fib1 += fib0; fib0 = tmp;
      phiN *= phi; psiN *= psi;
  }

  NT res = NT(1)/sqrt5 * (phiN-psiN);
  RA_ASSERT(fib1 == res);
}
```

(a) Computation of the Fibonacci numbers

```
template <class NT>
void binomial_theorem(const int n){
  NT x = sqrt(NT(13));
  NT y = sqrt(NT(17));
  NT xpy = x+y; NT xpyn = 1;

  NT *xi = new NT[n+1]; xi[0] = 1;
  for (int i = 1; i <= n; ++i) {
    xpyn *= xpy; xi[i] = xi[i-1] * x;
  }
  NT yi = 1; NT nchsi = 1;
  NT res = xi[n];
  for (int i = 1; i <= n; ++i) {
    nchsi *= NT(n-i+1)/NT(i);
    yi *= y; res += nchsi*xi[n-i]*yi;
  }
  RA_ASSERT(xpyn == res); delete[] xi;
}
```

(b) Computation of the binomial theorem

Figure 4.54: Source code in C++ for the Fibonacci test and the binomial theorem test. The template parameter `NT` specifies the used number type. The function `RA_ASSERT` verifies the passed boolean expression. Additional statements used for testing purposes in the original code are omitted.

Both tests lead to expression dags with interesting properties. The structure in each case is largely unbalanced due to the computation by the loops. At the same time, the expression dags contain a large number of blocking nodes. In the binomial theorem test, the expression for $\binom{n}{i}$ is largely composed of divisions. On the contrary, all operands occurring in the iterative formula for the Fibonacci numbers are integers.

The results for the Fibonacci test are shown in Figure 4.55. Both error bound balancing and restructuring reduce the running time of the computation. The largest differences, however, occur when either of the methods is able to preclude another iteration of the floating-point filter. As evident from the performance of AM-Balancing, the biggest part

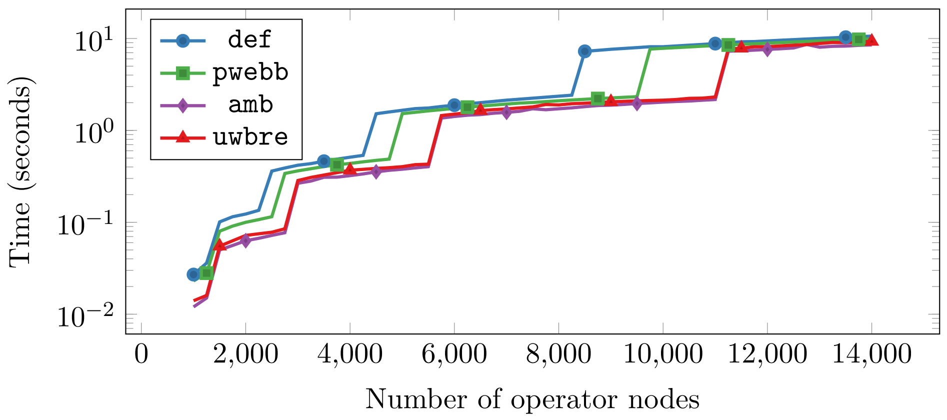


Figure 4.55: Running times of the Fibonacci test (logarithmic). At several points, the running time jumps when requiring an additional evaluation step. Aside from these jumps, error bound balancing reduces the running time by 5–22 % and Brent Restructuring by 12–55 %. AM-Balancing is more effective than the other strategies with a total reduction of 20–58 %. Near the jumps, improvements of up to 80 % can be observed.

of the running time improvement, including the decrease in the number of iterations, stems from the balancing of the large products used in the computations of ϕ^n and ψ^n. The reduction in the number of iteration steps is not caused by an improvement in the accuracy-driven evaluation, but by an improvement in the initial approximation and therefore by a shift of the starting point for the floating-point filter iteration. Somewhat counterintuitively, the configuration for the error bound balancing reduces the initial approximation as well, although error bound balancing is not active during the initialization step. This is a side effect of the more precise error representation used for error bound balancing. In Chapter 6, we elaborate on this effect. As we will see, the reduction of the initial accuracy is less an accomplishment of the balancing strategies than a hint on a suboptimal filter strategy in `Real_algebraic`.

Running times for the binomial theorem test are shown in Figure 4.56. Similar to the previously discussed experiments shown in Figure 4.49 and 4.50, Brent Restructuring strongly reduces the running time of the evaluation. Note that AM-Balancing only mildly reduces the running time, therefore the impact of balancing the computation of $(x + y)^n$ is small. In contrast to the previous experiments, however, Brent Restructuring not only leads to more bigfloat conversions but also improves the overall separation bound significantly, leading to a lower number of iterations of the floating-point filter. The divisions used for computing $\binom{n}{i}$ have small integer operands. If the numerator is enlarged, the probability increases that the result of the division is an integer or, at least, exactly representable by a floating-point number. By pushing divisions to the top, some of the denominators cancel out and thereby the size of the separation bound's magnitude is reduced. Paradoxically, whether this leads to a noticeable improvement largely depends on the size of x and y. In our example, x and y are of the form $x = \sqrt{x'}$ and $y = \sqrt{y'}$. If the operands inside the roots are smaller than 0.5, then the separation

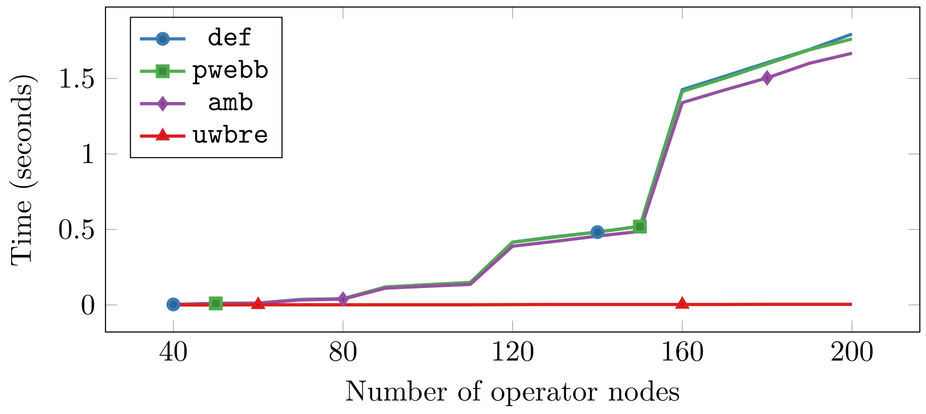


Figure 4.56: Running times of the binomial theorem test. Except for Brent Restructuring, the modified strategies show a mild decrease in running time ranging from around 1 % with error bound balancing to around 7 % with AM-Balancing. Brent Restructuring drastically reduces the running time. For $n = 200$, `uwbre` is about three decimal orders of magnitude faster than the default configuration.

bound improvement diminishes. Table 4.57 shows running times and separation bounds for various values of x' and y'. When one or both of the operands are decreased from 0.5 to 0.49 the separation bound suddenly becomes very small and, in consequence, the running times experience a steep increase. At a value of 0.5, the BFMSS[2] separation bound algorithm changes the strategy how it treats the root operation. Let v, u, l denote the values from the algorithm as shown in Table 2.7 (page 26). For 0.5, we have $2^v u = 1$ after the initialization. For smaller values, $2^v u < 1 = l$. Consequently, if one of x', y' is smaller than 0.5, the associated root operation yields a value for l that is larger than one, which leads to a fast growth of the parameters. The condition $2^v u \geq l$, which determines when to switch between the two root cases, originates in the BFMSS bound where an

x', y'		13/17	0.5/0.5	0.49/0.5	0.49/0.49
`def`	Time (s)	0.134	0.293	2.199	4.090
	Bound (log)	−83739	−123739	−649139	−1164139
`uwbre`	Time (s)	0.001	0.040	2.046	3.713
	Bound (log)	−1224	−41224	−566624	−1081624
`def`*	Time (s)	0.134	0.113	0.136	0.121
	Bound (log)	−83739	−83239	−93239	−93239
`uwbre`*	Time (s)	0.001	0.001	0.008	0.007
	Bound (log)	−1224	−724	−10724	−10724

Table 4.57: Running times and final separation bounds (magnitude) for the binomial theorem test with $n = 100$ with respect to different choices of $x = \sqrt{x'}$ and $y = \sqrt{y'}$. Data points are given for the default configuration and for the configuration with Brent Restructuring when used with the original and the modified BFMSS[2] separation bound (marked by a star).

equivalent condition $U \geq L$ is imposed on the algorithm (cf. [PY06]). To the author's knowledge, all current implementations use this condition [Bur+04; PY06; Mör15a]. Nevertheless, the choice of the condition is arbitrary in the sense, that both cases are always sufficient to ensure correctness. While in the BFMSS bound the parameters U and L are exponentiated in the respective cases, in the BFMSS[2] bound the exponentiation is applied to u and l. It might therefore be worthwhile to use the condition $u \geq l$ instead. In the lower half of Table 4.57, the running times and separation bounds for the binomial theorem test are shown with the modified separation bound. For these configurations, the paradoxical effect almost completely vanishes.

While still somewhat artificial, the Fibonacci test and the binomial theorem test demonstrate how bad structure can emerge in intuitively written programs. While the unbalanced product in the code for the Fibonacci numbers is relatively easy to spot for an experienced programmer, recognizing optimizations like the one achieved by restructuring in the binomial theorem test is much harder and requires deep knowledge of the underlying number type. Embedding optimization mechanisms into the evaluation process enables the programmer to write code more intuitively and moves exact number types one step closer to the ideal of an easily exchangeable general purpose number type.

5 Exact Decisions in a Parallel Environment

If separation bounds get small in complicated or large expressions, a high accuracy is needed to decide that a value is zero. The techniques presented in the previous chapter are designed to reduce the cost increase due to bad structure but they cannot reduce the cost below the base cost inflicted by the target accuracy, except for occasional bigfloat conversions. If the operations generally have high cost, parallelization turns out to be a viable technique to complement the balancing strategies [Wil18c]. In this chapter, we introduce and evaluate techniques to parallelize the evaluation of an exact-decisions number type on multiple processors. The first section of the chapter extends the theoretical framework of Section 4.1 to a parallel environment and describes basic paralellization strategies. In particular, it discusses how a limited number of processors should be assigned to evaluation tasks in an arithmetic expression dag in order to use them to their full potential. In Section 5.2, the techniques from the previous chapter are evaluated with respect to the parallelizability and partly adjusted to the new setting.

5.1 Parallel Evaluation Strategies

The evaluation of an expression dag can be divided into three parts: the initialization, the error propagation and the bigfloat evaluation. In the initialization phase, the initial approximations are computed and restructuring algorithms can be applied. During the error propagation, an error distribution is computed and requested accuracies are assigned to each node. Finally, in the last phase, a bigfloat approximation for each node is computed, such that the requested accuracy bound is satisfied. Technically, in each of those phases parallelization is possible. However, most of the time needed for the evaluation can be attributed to the bigfloat evaluation. Therefore, we consider multithreading only for the third phase. In a recursive evaluation scheme, the error propagation and the bigfloat evaluation phases intertwine in a way that is not suitable for parallelization (cf. Section 3.2.2). Consequently, as a starting point for parallel evaluation strategies we require the number type to use topological evaluation.

5.1.1 Parallel Evaluation Cost

Let E be an expression dag with n operator nodes. The evaluation of an operator node is dependent on the evaluation of its child nodes. If we assign one (virtual) processor to each operator node, the computation of the approximation can be started as soon as the computation at its child nodes is finished. The total evaluation time then depends on the most expensive path in E.

Definition 5.1. Let E be an expression dag with error distribution q. We denote the set of all root paths to all operator nodes in E as $\mathcal{P}(E) = \bigcup_{v \in \mathbf{V}(E)} \mathcal{P}(v)$. For a path $P \in \mathcal{P}(E)$, we call

$\mathcal{P}(E)$

$\mathrm{cost_t}(P)$

$$\mathrm{cost_t}(P) = \sum_{v \in \mathbf{V}(E)} p(v) \tag{5.1}$$

true cost

critical path

the *true cost* of P, where $p(v)$ denotes the operator precision at v. We call a path $P_c \in \mathcal{P}(E)$ with maximum true cost among the paths in $\mathcal{P}(E)$ a *critical path* in E.

For an arbitrary number of processors, the evaluation time of E directly depends on the true cost of the critical path in E. This is reflected in the following definition.

Definition 5.2. Let E be an expression dag with error distribution q. Then we call

$\mathrm{cost_p}(E)$

$$\mathrm{cost_p}(E) = \max_{P \in \mathcal{P}(E)} \mathrm{cost_t}(P)$$

parallel cost

the *parallel cost* of E.

With the standard error distribution, it is reasonable to assume that the critical path is one of the longest paths in E. The required accuracy and therefore the operator precision at a node tendentially increase with a higher depth of the respective node in E. Consequently, a higher number of summands in (5.1) tendentially leads to a larger sum and a longer path tendentially has higher true cost than a shorter one. Meanwhile, the cost of a path grows linearly in the target accuracy with a factor depending on its number of nodes. If a path is longer than another one, then with increasing target accuracy the cost of the longer path eventually surpasses the cost of the shorter one.

Observation 5.3. Let E be an expression dag with an error distribution q for which the accuracy increase $i(P)$ along any path P in E does not depend on the target accuracy $q(E)$. Then there is a threshold $t \in \mathbb{R}$, such that $q(E) \leq t$ implies that each critical path in E is a longest path in E.

The observation is an indication that in order to estimate the parallel evaluation cost it is reasonable to focus on the highest cost among the longest paths in E. In particular, it signifies the importance of the depth of an expression dag for parallel evaluation.

5.1.2 Implementation

In this section, we briefly highlight the main aspects for a parallel adaptation of exact-decisions number types. The descriptions are based on `Real_algebraic`, but can be expected to transfer to other number types as well. Details regarding the implementation of multithreading in `Real_algebraic` can be found in [Wil18a].

Shared Data

In order to preclude data corruption, access to shared data between threads must be protected or made unnecessary. Potential sources for shared data are the separation bound computation and the reference management, which may both be triggered after an evaluation. If the computed approximation is close to zero, a separation bound is computed to check whether it can be verified that the value is exactly zero. For the computation, an upper bound to the algebraic degree of the node's subexpression is determined by recursively traversing its descendants. To ensure that each descendant is visited at most once, a shared flag is used. Finally, if the value of the subexpression is zero, the node gets converted into a single bigfloat node, thereby deleting the references to its child nodes. In order to minimize the overhead of the parallelization, lock states associated with semaphores should be avoided whenever possible. For the reference counter management, *atomic* operations can be used instead, which take advantage of processor-level locks. The computation of the algebraic degree bound on the other hand can be changed to an iterative computation as described in Section 3.2.3, eliminating the need for write access to the child node data.

Dependency Management

For the parallelization, we assign an evaluation task to each node in the expression dag. The evaluation of an operator node must wait until the child tasks are completed. Using a semaphore-based notification system leads to a significant overhead, overshadowing most of the gains from the parallelization. Instead, we use a lock-free approach, where each node knows, and takes responsibility for, its parents. Each node gets assigned an atomic dependency counter initialized with the number of its child nodes. When the evaluation of a node is finished, the node reduces the number of dependencies of all of its parents by one. If a parent's dependency counter reaches zero, it starts the evaluation task for the parent. With this strategy, each task can in principal be executed immediately after it is created and therefore waiting states are avoided entirely. Note that this strategy requires shared access to the dependency counter for all child nodes. However, with a careful implementation it is sufficient to rely completely on the atomic operations of the dependency counter (cf. [Wil18a]).

Task and Thread Management

For the execution of evaluation tasks, each task must be assigned a thread. In customary multi-core processors, the number of simultaneously executable threads is currently in the single digit up to low double digit range. Creating more threads than supported by the processor leads to performance loss due to repeated context switches by the scheduler. The usage of a thread pool is a natural choice to limit the number of threads that can be active simultaneously. We use a simple thread pool based on a task queue. When a task is ready to start, we insert it into a queue. As long as the number of active threads is below a certain threshold and the number of remaining tasks in the queue is above a certain threshold, a new thread is created. Afterwards, each thread takes tasks from the queue until it is empty, at which point the thread gets terminated. The `boost` library provides a lockfree queue, which can be utilized for the task management under the condition that the the main algorithm is adjusted to fulfill certain properties. Most importantly, with a lockfree implementation there is no safe way to determine the current size of the queue. Therefore, the main evaluation algorithm must be able to decide locally whether its computation has finished, i.e., whether all nodes have been evaluated. Since each node knows its parents, this can be determined by a check whether the current node does not have any parents and therefore is the root node of the expression dag.

Depth Prioritization

In Section 5.1.1, it was observed that the running time of a parallel evaluation with arbitrary many processors depends on the critical path in an expression dag. If the number of processors is limited, the evaluation time is bounded from below by n/p, where n is the total (true) cost of the evaluation and p is the number of processors available. In this case, the evaluation time may depend on the order in which tasks are processed.

Assume that for an expression dag E each operation takes exactly the same time. If a first-in, first-out queue is used for task management, the operations at the nodes are processed in a bottom-up, breadth-first manner. That is, if the subexpression at a node u has a smaller depth than the subexpression at a node v, then u is evaluated before v. This behavior may lead to a suboptimal use of the available processors if shallow parts of the expression dag are evaluated before long paths and the remaining structure does not allow a parallel evaluation.

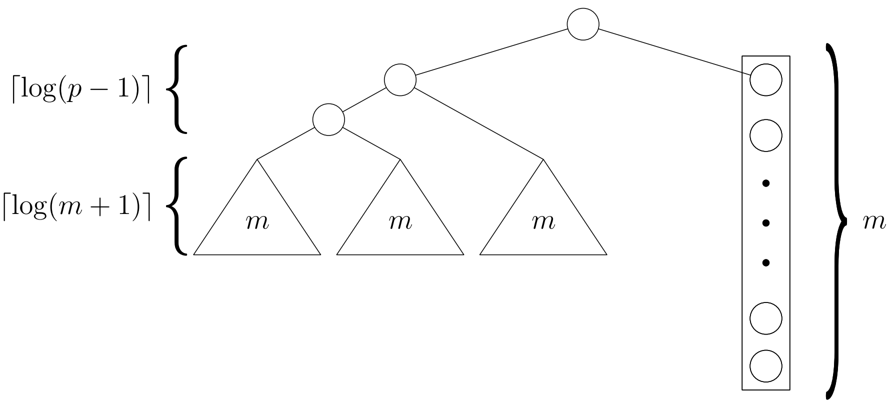


Figure 5.1: Schematic depiction of the expression dag $E_{m,p}$ for $p = 4$.

$E_{m,p}$ For $m, p \in \mathbb{N}$ let $E_{m,p}$ be an expression dag, such that (cf. Figure 5.1)

1. the left child of the root is a balanced expression dag T_0 with $p-1$ operands, each of which is a balanced expression dag T_i, $1 \le i \le p-1$, with m operator nodes, and
2. the right child of the root is a list-like expression dag T_{list} with m operator nodes.

For simplicity, assume that the evaluation of any operator node in E takes time 1. Then $E_{m,p}$ can be evaluated in time $m + \lceil \log(p-1) \rceil + 1$ with p processors by employing the following task order:

- During the first m steps, assign one processor each to $T_1, ..., T_{p-1}$ and one processor to T_{list}.
- During the remaining steps, assign all processors to the evaluation of T_0 and the root node.

Evaluating $E_{m,p}$ with a first-in, first-out task order takes almost twice the time as the above evaluation strategy for large p. The balanced trees $T_1, ..., T_{p-1}$ have depth $\lceil \log(m+1) \rceil$. Due to the bottom-up, breadth-first processing order, they are evaluated first, together with the bottom $\lceil \log(m+1) \rceil$ nodes of T_{list}. This evaluation takes time

$$\left\lceil \frac{m(p-1) + \lceil \log(m+1) \rceil}{p} \right\rceil \ \geq \ m - \left\lceil \frac{m}{p} \right\rceil$$

Afterwards, T_0 and the remaining part of T_{list} is evaluated together with the root node. From this point on, at each time at most p tasks are simultaneously ready. Since T_{list} must be evaluated in sequence, the evaluation takes time $m - \lceil \log(m+1) \rceil + 1$. Altogether, the total evaluation time is at least

$$m - \left\lceil \frac{m}{p} \right\rceil + m - \lceil \log(m+1) \rceil + 1 \ \geq \ 2m - \left\lceil \frac{m}{p} \right\rceil - \lceil \log(m+1) \rceil \tag{5.2}$$

The task order can be improved by adding priorities to the evaluation tasks. A simple but effective prioritization is achieved by considering the depth of a node. If a higher maximum distance to the root leads to faster processing, an optimal usage of the available processors is guaranteed for the evaluation of $E_{m,p}$. This generalizes to every evaluation in which a longer path is at least as expensive as a shorter one, which, for example, is always the case for high target accuracies (cf. Observation 5.3). We call this task evaluation strategy *depth prioritization*. Implementing a task manager with depth prioritization naturally requires replacing the simple queue used in the first-in, first-out approach with a priority queue. For the implementation of the priority queue, we fall back to a lock-based approach. While efficient algorithms for lock-free priority queues have been proposed [ST05], there are no available implementations in standard libraries. Due to the relative complexity of implementing such a queue, a lock-based implementation is deemed to be sufficient.

depth prioritization

5.1.3 Experimental Evaluation

We compare two configurations that extend the default configuration (Figure 4.30, page 110) by a multithreaded evaluation with and without depth-prioritized task management. The two configurations are depicted in Figure 5.2. Our experimental setup is identical to the one presented in Section 4.4.1. Note that the processor of the test machine contains four cores with eight threads. Hence, a speedup of four is the maximum

defm : def

TaskPolicy:
`First_in_first_out_task_policy`
OperationComputationPolicy:
`Parallel_operation_computation`
ReferenceCounterPolicy:
`Mt_safe_reference_counter`

defmp : def

TaskPolicy:
`Depth_prioritized_task_policy`
OperationComputationPolicy:
`Parallel_operation_computation`
ReferenceCounterPolicy:
`Mt_safe_reference_counter`

Figure 5.2: Configurations for the default number type with a multithreaded evaluation. Tasks are prioritized either first-in, first-out with a lockfree queue (`defm`) or prioritized by depth (`defmp`).

speedup achievable through multithreading. For both task managers we use at most four threads and a minimum number of five waiting tasks before a new thread is created.

Multithreading has the highest impact if the expression dag is balanced. Furthermore, it is more useful if the individual bigfloat operations are more expensive. In a balanced or almost balanced expression dag, the depth is logarithmic in the size of the dag and therefore the evaluation cost of single nodes grows at most logarithmically along their root paths. Therefore, a higher number of operator nodes does not have a significant impact on the achieved speedup. Instead, the target accuracy for the evaluation is of high relevance since a higher target accuracy equally affects the cost of each operation. Figure 5.3 shows the evaluation cost of a balanced expression dag with 20000 operator nodes with uniformly randomly distributed operators as in Section 4.4 with respect to the requested target accuracy. For low target accuracies up to $z = -2000$, multithreading

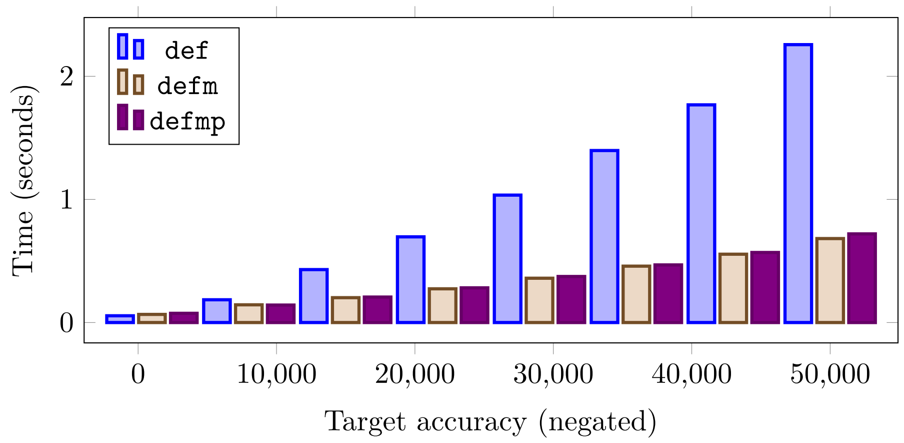


Figure 5.3: Running times for multithreading strategies for balanced expression dags with 20000 operator nodes and random operators. Data points are shown for a total accuracy of $z = -1000$ to $z = -50000$ in steps of -7000. For a high target accuracy, multithreading on a quadcore processor achieves a speedup of 3.3. For low target accuracies, the running time is slightly increased. A change from the first-in, first-out strategy to depth prioritization shows no improvement.

is more expensive than a single-threaded evaluation. If the target accuracy is high, a speedup of 3.3 is achieved. Since for perfectly balanced expression trees the first-in, first-out task prioritization already leads to a perfect evaluation order, depth prioritization is useless and leads to a small overhead of about 5 % compared to the unprioritized task management.

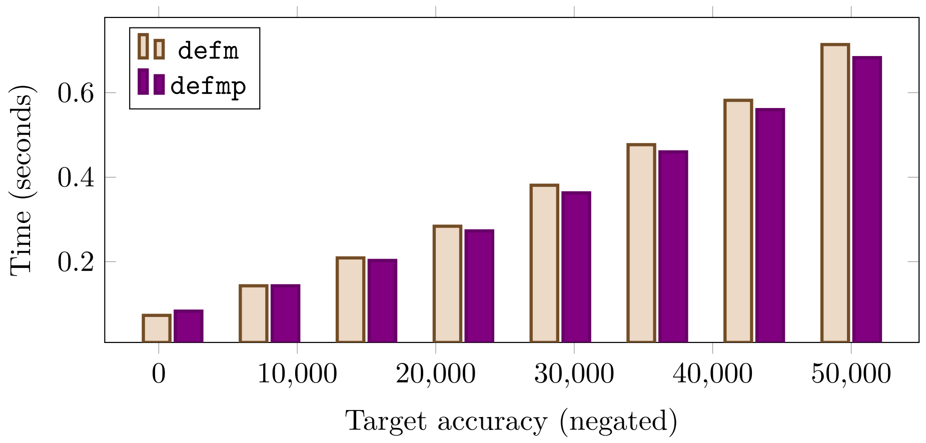


Figure 5.4: Running times for multithreading strategies for random expression dags with 20000 operator nodes and random operators. Prioritizing the tasks by their depth slightly improves the overall running time.

If a randomly created expression tree is used instead of a perfectly balanced one, depth prioritization gets a small advantage. Since randomly generated trees usually resemble balanced variants, the speedup compared to single-threading is similar to before. However, occasional long runs can lead to an unfortunate evaluation order. By prioritizing the tasks by their depth, the full potential of multithreading can be used up until the last few nodes. In Figure 5.4 we can observe that, on average, depth prioritization makes the evaluation of randomly generated trees about 4 % faster, except for low target accuracies where the evaluation order is largely irrelevant since multithreading does not improve on the singlethreaded running time. A larger difference between the task management policies can be expected when evaluating the expression dag $E_{m,p}$ introduced in Section 5.1.2, consisting of $p - 1$ balanced trees and a list-like expression dag with m operator nodes each. Figure 5.5 shows the running time for $m = 5000$ and $p = 4$ with random operators. It is evident that the depth prioritization leads to a significantly faster evaluation, taking roughly 65 % of the time of the first-in, first-out approach for large target accuracies. For larger accuracies it can be shown that this proportion goes down to about 60 %, as we would expect from the estimate in (5.2). If the structure of the expression dag leads to a high number of dependencies, the performance difference between single- and multithreaded evaluation vanishes. In a list-like expression dag, each node must be evaluated in sequence, therefore there is no benefit in having more than one thread. In fact, the task manager destroys threads that do not have a task assigned and therefore ensures that only one thread is active during the evaluation of a list of operator nodes. In Figure 5.6, the running times for a list-like expression dag are shown. A small gain

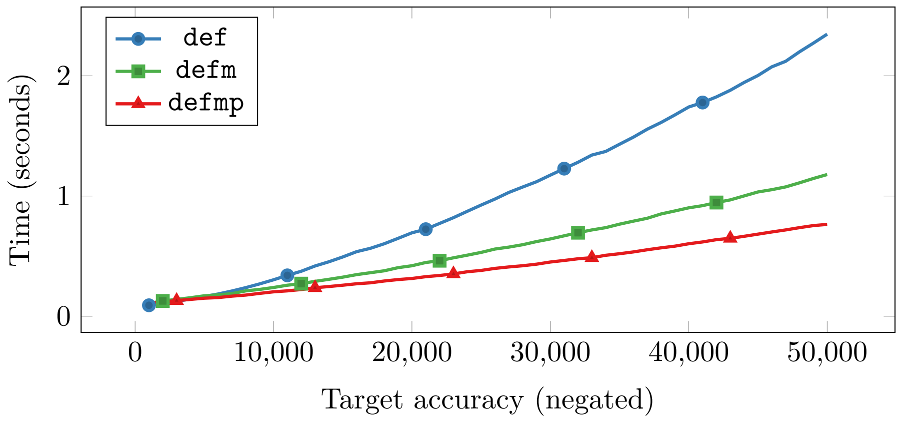


Figure 5.5: Running times for multithreading strategies for the expression dag $E_{m,p}$ with $m = 5000$, $p = 4$ and random operators. For high target accuracies, depth prioritization increases the speedup from 2.0 to 3.0 compared to the first-in, first-out approach.

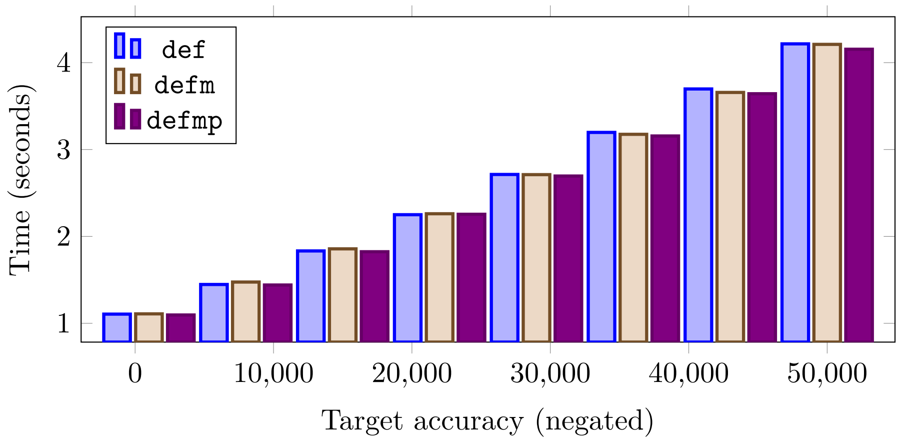


Figure 5.6: Running times for multithreading strategies for list-like expression dags with 20000 operator nodes and random operators. Multithreading shows almost no effect. Prioritizing the tasks by their depth improves the overall running time by about 1 %.

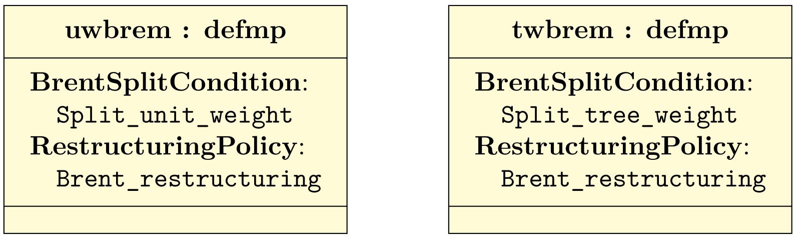
in running time can be observed due to a parallel evaluation of the divisions used in the operands. Depth prioritization leads to a slight improvement by assigning priority to the critical path, whereas the first-in, first-out evaluation does not start to evaluate the operator nodes in the list at all until all operator nodes in the operands have been evaluated.

5.2 Dependency Reduction

In the previous section, it was shown that parallelization is a viable strategy to reduce the running time of exact-decisions number types if there is low dependence between the nodes in the underlying graph structure. In this section, we evaluate how the parallelizability of an expression dag can be improved using the techniques established in Chapter 4.

5.2.1 Restructuring in a Parallel Context

Reducing the depth of an expression dag generally leads to a reduction on the number of dependencies. In Section 5.1.1, a cost model based on critical paths has been established. The restructuring methods from Section 4.3 can be applied to reduce the parallel cost by reducing the depth of an expression dag. In fact, Brent's restructuring method was originally developed for improving arithmetic expressions for a parallel evaluation. In this section, we experimentally evaluate the impact of restructuring on the parallelizability of an expression dag. For that, we use configurations with depth-prioritized multithreading as shown in Figure 5.7, which additionally implement either Brent Restructuring with unit weights or Brent Restructuring with the tree weight heuristic.



uwbrem : defmp

BrentSplitCondition:
`Split_unit_weight`
RestructuringPolicy:
`Brent_restructuring`

twbrem : defmp

BrentSplitCondition:
`Split_tree_weight`
RestructuringPolicy:
`Brent_restructuring`

Figure 5.7: Configurations for a multithreaded number type with restructuring strategies, implementing either unit weight (`uwbrem`) or tree weight (`twbrem`) Brent Restructuring.

The increase in parallelizability through restructuring is demonstrated for a list-like expression dag in Figure 5.8. As shown before, making use of multiple processors without additional effort is not possible in this case. With restructuring, in addition to the reduction in single-threaded running time, parallelization provides a speedup of 3.35, which is comparable to the speedup achieved for a perfectly balanced expression tree (cf. Figure 5.3). However, a speedup increase through restructuring is only possible if there are large parts of the expression dag where less nodes are ready for an evaluation than processors are available. For a small number of processors, this usually requires one large list-like subgraph, which either contains most of the operator nodes or fully separates one part of the expression dag from another. Let $E_{list}^{(k)}$ be an expression dag consisting of k equally sized list-like expression dags, put together by a balanced operator tree. Figure 5.9 shows the parallelization speedups for the default configuration and the configuration using Brent Restructuring during the evaluation of $E_{list}^{(k)}$ with 20000 operands for $1 \leq k \leq 4$. Since restructuring always leads to the same speedup, it is only depicted once. For less than four lists, restructuring can improve the parallelizability of the structure. After that, restructuring slightly decreases the speedup. Note that for

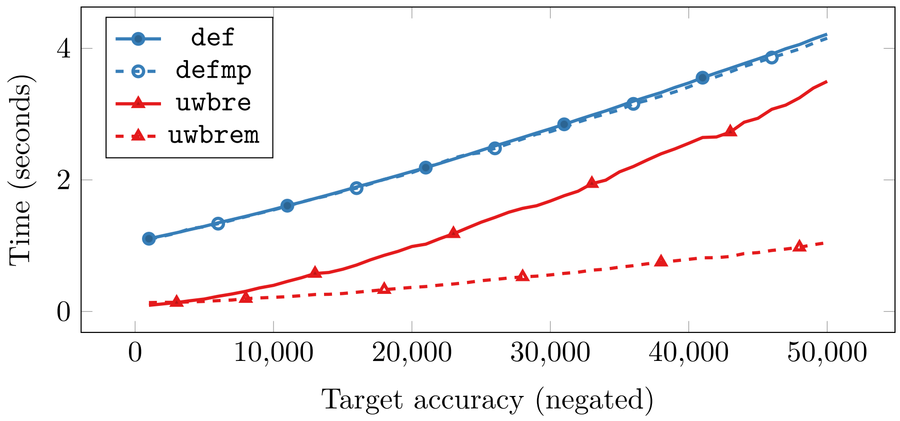


Figure 5.8: Parallel evaluation times with and without Brent Restructuring for a list-like expression dag with 20000 operator nodes and random operators. For `def`, multithreading shows no effect. After restructuring, the parallel evaluation time is decreased by 75–88 %. The relative gain decreases slightly with increasing target accuracy.

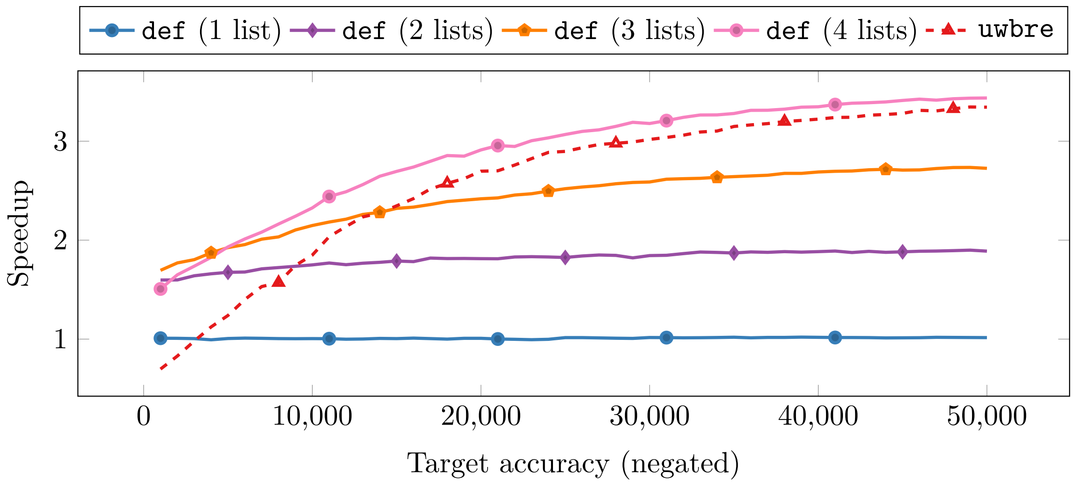


Figure 5.9: Speedups through multithreading for the evaluation of expression dags with 20000 operator nodes and random operators, consisting of one to four equally sized list-like expression dags. The maximum speedup of the default configuration is 1.0, 1.9, 2.7 and 3.4, respectively. After restructuring a maximum speedup of 3.35 is achieved for each of these expression dags with a nearly identical progression.

small target accuracies the costs of single bigfloat operations are too low to achieve a large speedup. For the default configuration, there is still a significant speedup due to the cost increase caused by the bad structure. A high target accuracy on the other hand negatively affects the relative performance of configurations using Brent Restructuring by increasing the impact of the additional operations (cf. Figure 4.40, page 116). For $E_{list}^{(4)}$, these effects lead to a net increase in parallel evaluation time when Brent Restructuring

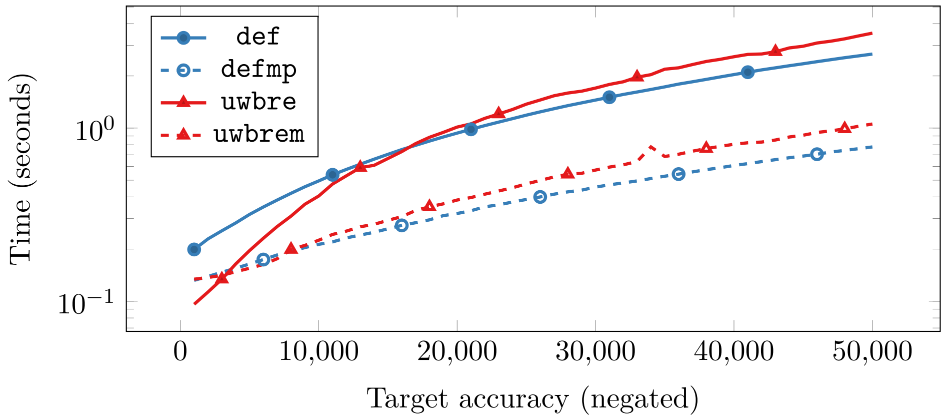


Figure 5.10: Parallel evaluation times before and after the application of Brent Restructuring for an expression dag consisting of four list-like expression dags with 5000 operator nodes each and random operators. Between a target accuracy of −2000 and −7000, the configuration using restructuring is around 5 % faster than the default configuration if multithreading is enabled. For $z = -1000$ it is slightly slower. For the displayed data points with $z \leq -8000$, `uwbrem` is 3–35 % slower than `defmp` (note the logarithmic scale).

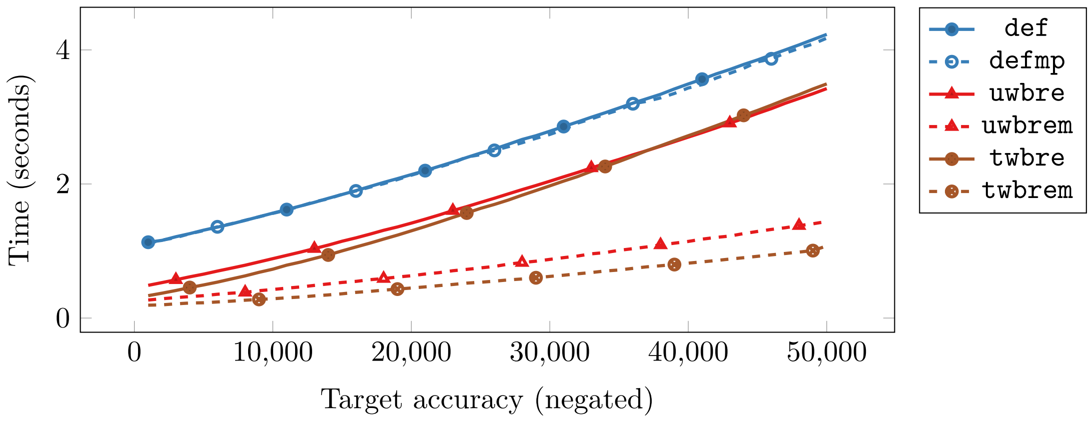


Figure 5.11: Parallel evaluation times for Brent Restructuring with and without weights for a list-like expression dag with 20000 operator nodes, random operators and 10 % nodes with an additional external reference. The sequential running times after restructuring are similar for unit weights and for tree weights. The parallel evaluation time after restructuring is 30 % lower if tree weights are used instead of unit weights.

is enabled for both small and large target accuracies as shown in Figure 5.10.

If blocking nodes are present, Brent Restructuring cannot create an optimal structure. Still, for a small number of blocking nodes, the full speedup can be maintained. Figure 5.11 shows evaluation times for a list-like expression dag with random operators and 10 % blocking nodes. Both with unit weights and with the weighted approach based on the tree weight, restructuring significantly improves on the parallel evaluation time. By weighting operands, the speedup can be increased from about 2.4 to the presumably optimal speedup of 3.4 for the highest target accuracy. Weighted Brent Restructuring is able to largely isolate the blocking nodes and balance the rest of the expression dag (cf. Theorem 4.78). If p processors are available, they can be fully utilized if, after restructuring, the total evaluation cost of the list of blocking nodes is equal to at most $p - 1$ times the evaluation cost of the then balanced rest of the expression dag. If the evaluation cost of each node would be equal, this would imply that an optimal speedup can be achieved if at most an $1/p$-th of the operator nodes are blocking. Since in our experimental setup we have $p = 4$, this equates to 25 % blocking nodes.

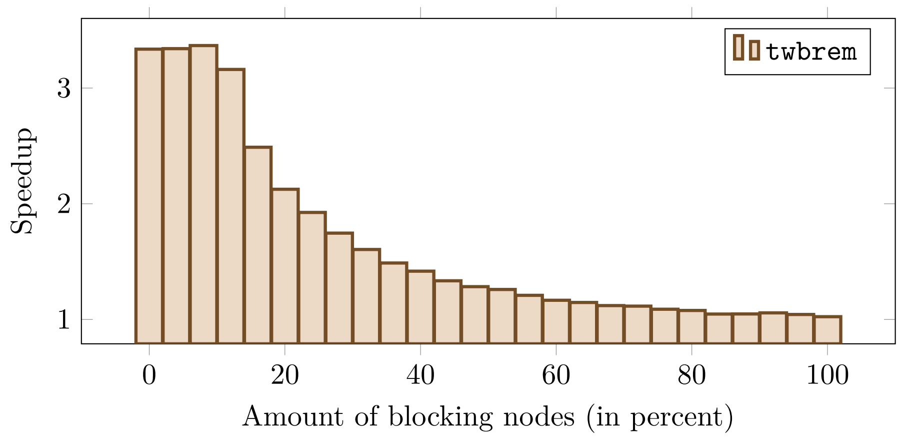


Figure 5.12: Speedup after Weighted Brent Restructuring with the tree weight for the evaluation to $z = -50000$ of a list-like expression dag with 20000 operator nodes, random operators and varying degrees of blocking nodes. With up to 10 % blocking nodes, the maximum speedup is achieved. Starting from around 15 % blocking nodes, the speedup drops rapidly.

Figure 5.12 shows the speedup on a list-like expression dag with varying numbers of blocking nodes that was restructured by Weighted Brent Restructuring. We can observe a harsh drop starting at a rate of about 15 % blocking nodes. Although, in this case, less than a sixth of the original nodes are blocking, the longest path in the restructured graph consistently covers 26–29 % of its operator nodes. Beside the blocking nodes, the longest path usually contains more than twice as many additional nodes that are used to create Divide-Structures. Hence, the optimal speedup declines faster than expected through the previous estimation. With a close examination of the procedure `make_DS`, one comes to the conclusion that the longest path must always be at least twice as long as the number of blocking nodes.

5.2.2 Error Distributions for Multithreading

Error bound balancing, as introduced in Section 4.2, is condemned to have at most a minor impact on the parallelizability of an expression dag. Since dependencies cannot be changed by a modified error distribution, list-like expression dags cannot be made parallelizable through error bound balancing. Otherwise, if large parts of the structure are already balanced, the error distribution is already mostly balanced as well. Nevertheless, a carefully chosen error distribution may enhance the results achieved by restructuring in situations where a perfectly balanced structure cannot be accomplished.

The path weight error distribution minimizes the absolute cost of the evaluation of an expression dag, as proven in Theorem 4.33. If an expression dag is only partly balanced or not balanced at all, using this distribution can significantly reduce the single-threaded evaluation time. The optimality, however, does not transfer to a parallel environment. The path weight favors shallow large parts of an expression dag over deep parts with few nodes. The absolute cost of the deep parts is increased in order to decrease the absolute cost of the large parts, thereby increasing the total parallel cost. Furthermore, even for balanced expression dags, the impact of an accuracy increase at an inner node on the parallel evaluation cost is underrated since it is weighed against all operator nodes in its subexpression while the parallel evaluation cost increases only in proportion to the number of nodes on one (the critical) path. Consider an expression tree T with n operator nodes and depth k. For simplicity, assume that each operation is an addition and therefore each operator constant is equal to one. Furthermore, assume that the correction addend for converting a required accuracy to an operator precision is always equal to γ. Then with equation (4.12) from Lemma 4.31 (page 62), the accuracy increase along each root path is $-\log n$ and therefore the required accuracy at each operator node is equal to $z - \log n$, where z refers to the target accuracy of the evaluation. The critical path of T is therefore one of the longest paths in T and the parallel evaluation cost of T is $k \log n - kz + k\gamma$. However, the minimum true cost required for evaluating a list-like expression dag with k nodes is only of the order of $k \log k + k(\gamma - z)$. This difference leaves room for improvement.

Lemma 5.4. For $m, p \in \mathbb{N}$ let $E_{m,p}$ be an expression tree consisting of $p-1$ balanced trees and one list-like expression tree, each of size m as defined in Section 5.1.2. Let all operations be additions and all correction addends be equal to γ. Then for each $\varepsilon > 0$, there is a valid error distribution such that

$$\mathrm{cost_p}(E_{m,p}) = (m+1)\log(m+1) + \varepsilon + (m+1)(\gamma - z)$$

if m or $\gamma - z$ is sufficiently large.

Proof. Let v_0 be the root node of $E_{m,p}$ with outgoing edges e_l, e_r. Let q be an error distribution with

$$\begin{aligned} i(v_0) &= -\log(m+1) - \varepsilon \\ i(e_l) &= -\log(m+1) + \log(1 - 2^{-\varepsilon}) \\ i(e_r) &= \log\left(\frac{m}{m+1}\right) \end{aligned} \tag{5.3}$$

that is otherwise identical to the path weight error distribution. The required accuracies for the root node fulfill the balancing constraint (4.8) and therefore q is valid. Let P_l be a path with the highest true cost in $E_{m,p}$ among the paths using e_l and let P_r be a path with the highest true cost in $E_{m,p}$ among the paths using e_r. Then P_l contains $k = 1 + \lceil \log(p-1) \rceil + \lceil \log(m+1) \rceil$ operator nodes and, since the left subexpression of $E_{m,p}$ contains $r = m(p-1) + p - 2$ operator nodes, it has true cost

$$\mathrm{cost_t}(P_l) = k \log r + k \log(m+1) - (k-1)\log(1 - 2^{-\varepsilon}) + \varepsilon + k(\gamma - z)$$

On the other hand, P_r contains $m+1$ operator nodes and its true cost is given as

$$\begin{aligned}\mathrm{cost_t}(P_r) &= m \log m - m \log\left(\frac{m}{m+1}\right) + \log(m+1) + \varepsilon + (m+1)(\gamma - z)\\ &= (m+1)\log(m+1) + \varepsilon + (m+1)(\gamma - z)\end{aligned}$$

For large m or, as long as $m + 1 > k$, for large $\gamma - z$, the cost of evaluating P_r is higher than the cost of evaluating P_l. Therefore, $\mathrm{cost_p}(E_{m,p}) = \mathrm{cost_t}(P_r)$ and the lemma is proven. □

Lemma 5.4 suggests that there are situations in which the parallel cost can be reduced by a significant amount when using an alternate error distribution. In an optimal error distribution for parallel evaluation, each path in the expression dag must be a critical path. Otherwise, we can use a similar argument as in Lemma 4.26 to show that we can reduce the cost of the other critical paths. Finding such an error distribution requires solving equations of the form

$$ax^b + cx - 1 = 0$$

with arbitrary $a, b, c \in \mathbb{Q}$ (see [GW19] for details). In general, a solution for the root of polynomials of arbitrary degree is not representable in a closed form. Therefore, one must fall back to heuristics or numerical methods when following this approach.

As described in Section 5.1.1, it is reasonable to assume that the critical path in an expression dag turns out to be one of its longest paths. For a heuristic, we can therefore focus on minimizing the cost of the longest paths. The distribution in (5.3), used for the root node in the proof of Lemma 5.4, provides some insight on how subexpressions with differing depths can be handled. By distributing the cost increase equally among both outgoing edges if both subexpressions of an operation have the same depth, we arrive at the following error distribution.

Definition 5.5. Let E be an expression dag. For a quasi-unary node $u \in \mathbf{V}(E)$ let e be its outgoing edge in $\mathbf{E}(E)$ and let d be the depth of the subexpression at the target node of e. For a fully binary node $v \in \mathbf{V}(E)$ let e_l, e_r be its outgoing edges and let d_l, d_r be the depth of the left and right subexpression of v. We define an error distribution q for any target accuracy z. Let $q(E) = z$ and let $i(u_0) = 0$ for each quasi-leaf u_0. Furthermore, for each quasi-unary node u let

$$\begin{aligned}i(u) &= -\log(d+1)\\ i(e) &= \log\left(\frac{d}{d+1}\right) - \log(c(e))\end{aligned}$$

and for each fully binary node v with $d_l > d_r$ (vice versa for $d_r > d_l$) let

$$\begin{aligned} i(v) &= -\log(d_l + 1) - 1 \\ i(e_l) &= \log\left(\frac{d_l}{d_l + 1}\right) - \log c_l \\ i(e_r) &= -\log(d_l + 1) - 1 - \log c_r \end{aligned}$$

Finally, for each fully binary node v with $d_l = d_r$ let

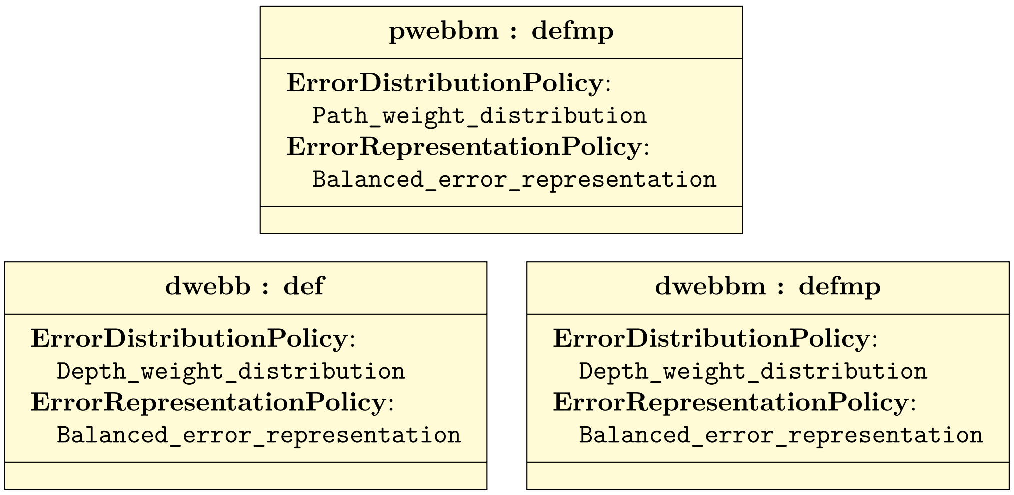
$$\begin{aligned} i(v) &= -\log(d_l + 1) \\ i(e_l) &= \log\left(\frac{d_l}{2(d_l + 1)}\right) - \log c_l \\ i(e_r) &= \log\left(\frac{d_l}{2(d_l + 1)}\right) - \log c_r \end{aligned}$$

Then we call the error distribution q resulting from this definition the *depth weight error distribution* for E and z.

depth weight error distribution

Observation 5.6. The depth weight error distribution is a valid error distribution.

In order to test the impact of error bound balancing on the parallelizability experimentally, in Figure 5.13 we introduce configurations using the path weight and the depth weight error distribution together with multithreading. As a reference point, we furthermore define a configuration with the depth weight error distribution but without multithreading. Let $E_{m,p}$ be an expression tree as in Lemma 5.4 with $p = 4$ and $m = 20000$. The standard distribution leads to a parallel evaluation cost that is



pwebbm : defmp

ErrorDistributionPolicy:
`Path_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`

dwebb : def

ErrorDistributionPolicy:
`Depth_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`

dwebbm : defmp

ErrorDistributionPolicy:
`Depth_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`

Figure 5.13: Configurations for a multithreaded number type with the path weight error distribution (`pwebbm`) and for both a single- and a multithreaded number type with the depth weight error distribution (`dwebb`, `dwebbm`).

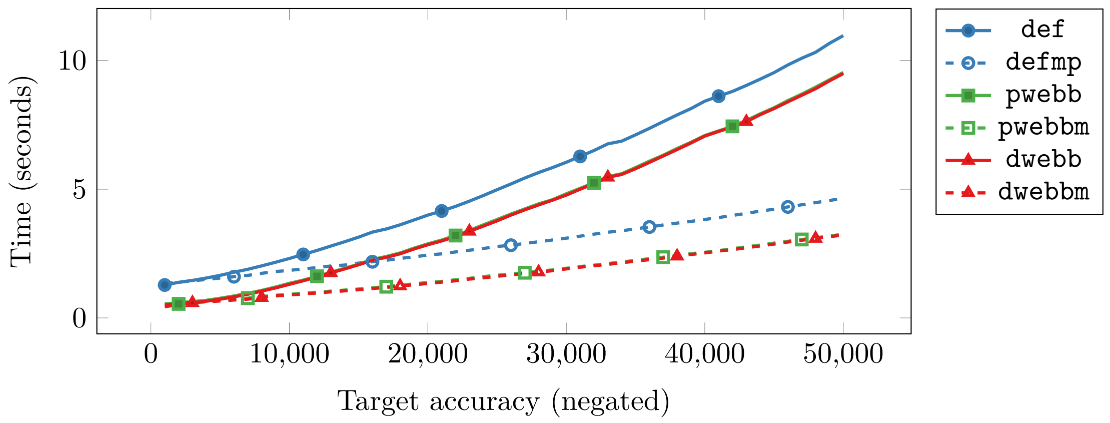


Figure 5.14: Parallel evaluation times of the expression dag $E_{m,p}$ with $p = 4$, $m = 20000$ and random operators for the path weight and depth weight error distributions. Both error bound balancing strategies reduce the parallel evaluation time by 30–60 % with a higher relative reduction for low target accuracies. Between the two strategies there is no significant difference.

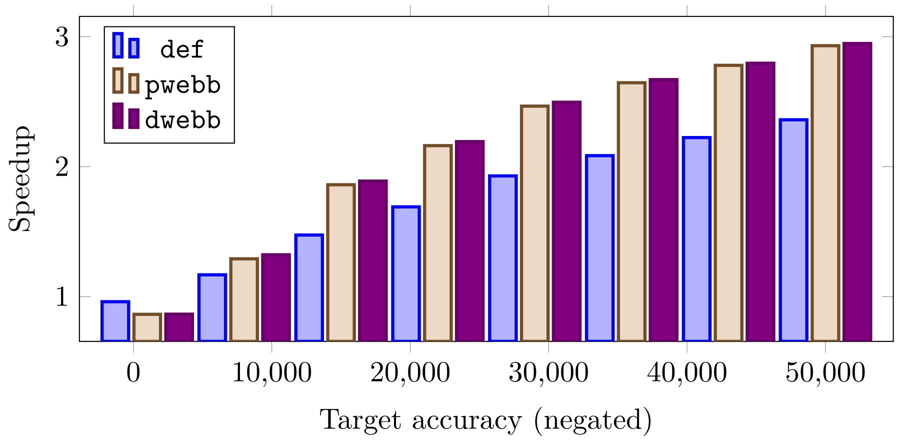


Figure 5.15: Speedup for the path weight and depth weight error distributions regarding the parallel evaluation of the expression dag $E_{m,p}$ with $p = 4$, $m = 20000$ and random operators. Starting from $|z| = 8000$, both error bound balancing increase the achieved speedup. The depth weight error distribution consistently leads to slightly higher speedups than the path weight error distribution.

quadratic in m. We could therefore expect an increase in parallelizability due to error bound balancing. Figure 5.14 compares the parallel evaluation time of $E_{m,p}$ with random operators with and without error bound balancing methods. Both error bound balancing methods decrease the running time of the evaluation. They lead to both a moderate decrease in single-threaded running time and an increased speedup (Figure 5.15). The depth heuristic achieves a slightly higher speedup than the path weight. A difference in the total evaluation time, however, is barely noticeable. Since $p = 4$, compared to the path weight error distribution, the size of the required accuracy along the longest path in

$E_{m,p}$ is decreased by two per node in the depth weight error distribution and therefore the parallel evaluation cost is reduced in total by $2m$. In relation to the cost induced by the operation constants and the target accuracy, this difference is insignificant. If the disparity between the number of nodes in the longest path and the total number of nodes grows, the difference in the parallel cost increases. For a small number of processors, however, the gain through the depth weight heuristic is always outweighed by the difference in the absolute cost between the two error distributions.

For a perfectly balanced tree, both subexpressions at each node have the same depth. Hence, there are no disparities between siblings that can be exploited by the depth heuristic. Nevertheless, the parallel evaluation cost of the tree is reduced by a shift in weight from the nodes to the edges.

Lemma 5.7. For $m \geq 1$ let E^m_{bal} be a perfectly balanced expression tree with $2^m - 1$ operator nodes. Let all operations be additions and all correction addends be equal to γ. Then the parallel cost for evaluating E^m_{bal} to accuracy z using the depth weight error distribution is

$$\mathrm{cost_p}(E^m_{bal}) = m \log m + \frac{m(m-1)}{2} + m(\gamma - z) \tag{5.4}$$

Proof. We prove the claim by induction on m. For $m = 1$, the single node is evaluated directly up to accuracy z, therefore the parallel cost is $\gamma - z$. For $m > 1$ let v_0 be the root node of E^m_{bal} with outgoing edges e_l, e_r. Since both of its children have depth $m - 1$, we are in the third case of Definition 5.5. Then $i(v_0) = -\log(m)$ and $i(e_l) = i(e_r) = \log(m-1) - \log(2m)$ and with the induction hypothesis we have

$$\begin{aligned}
\mathrm{cost_p}(E^m_{bal}) &= \mathrm{cost_p}(E^{m-1}_{bal}) - (m-1)\log\left(\frac{m-1}{2m}\right) + \log(m) + \gamma - z \\
&= (m-1)\log(m-1) + \frac{(m-1)(m-2)}{2} + (m-1)(\gamma - z) \\
&\quad - (m-1)\log\left(\frac{m-1}{2m}\right) + \log(m) + \gamma - z \\
&= (m-1)(\log m + 1) + \frac{(m-1)(m-2)}{2} + \log(m) + m(\gamma - z) \\
&= m \log m + \frac{m(m-1)}{2} + m(\gamma - z)
\end{aligned}$$

□

It can be shown that the cost increase at each step of the induction is the smallest increase possible and therefore the parallel cost in (5.4) is the smallest parallel cost achievable through the choice of a valid error distribution. Assume q is a valid error distribution leading to the optimal parallel cost of E^m_{bal}. Let $\delta(v_0), \delta(e_l), \delta(e_r)$ be the differences in the accuracy increases between q and the depth weight error distribution. Due to the symmetry, the cost increase at both child nodes must be the same in an optimal distribution, therefore $\delta(e_l) = \delta(e_r)$. With a similar argument as used in the proof of Lemma 4.32 on page 63 we can then conclude that

$$\delta(v_0) \leq -\frac{m-1}{2}\delta(e_l) - \frac{m-1}{2}\delta(e_r) = -(m-1)\delta(e_l) \tag{5.5}$$

Since an increase of e_l affects $(m-1)$ nodes, (5.5) must be an equality, which is only possible if $\delta(v_0) = \delta(e_l) = 0$. Hence, the depth heuristic leads to the optimal parallel cost for E^m_{bal}. With the path weight error distribution, the parallel evaluation cost of this expression tree is given as $m \log(2^m - 1) + m(\gamma - z)$, therefore the variable part of the cost is cut almost to half by the depth heuristic. Figure 5.16 shows the speedups

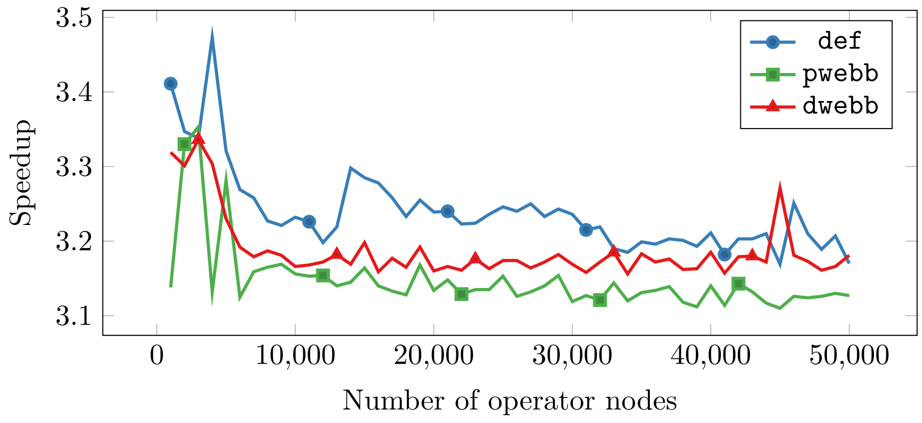
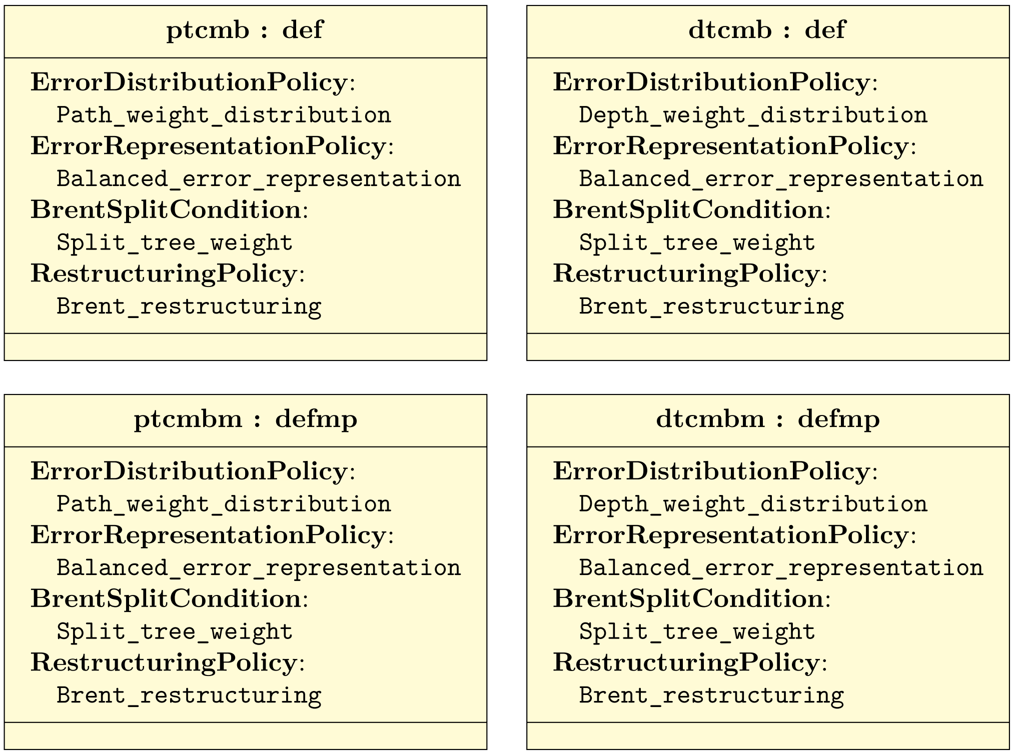


Figure 5.16: Speedup for various error distributions regarding the parallel evaluation of balanced expression dags with random operators to total accuracy −50000. The standard error distribution usually leads to the highest speedup. The speedups for the depth weight error distribution are slightly higher than for the path weight error distribution.

for evaluating a perfectly balanced tree with different numbers of operator nodes to accuracy −50000. There is no significant difference between the two error bound balancing methods. Interestingly, the standard error distribution consistently leads to a slightly higher speedup than the other two configurations. One possible explanation for this behavior might be that in the standard error distribution much weight is shifted toward the leaves where a high number of nodes can be evaluated simultaneously. Regarding the evaluation time, all three strategies perform remarkably similar with a maximum difference of 6 % of the smallest value for both single- and multithreaded evaluation time for all but one data points.

At the end of Section 4.4.2, it was shown that error bound balancing can be used to assist restructuring methods if blocking nodes are present (cf. Figure 4.47, page 121). In addition to a general performance increase, the parallelizability of partly restructured expressions may be increased by error bound balancing as well. In order to test for the speedup, we introduce four new configurations, combining the tree weight approach for Weighted Brent Restructuring with either the path weight or the depth weight error distribution for both serial and parallel evaluation (Figure 5.17). Running times for the combined methods in relation to pure Weighted Brent Restructuring in the presence of various percentages of blocking nodes are depicted in Figure 5.18. As expected from the previous experiments, error bound balancing reduces both the serial and the parallel running time after restructuring when blocking nodes are present. In contrast to the previous tests, the depth heuristic performs significantly worse than the path weight for a



ptcmb : def

ErrorDistributionPolicy:
`Path_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`
BrentSplitCondition:
`Split_tree_weight`
RestructuringPolicy:
`Brent_restructuring`

dtcmb : def

ErrorDistributionPolicy:
`Depth_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`
BrentSplitCondition:
`Split_tree_weight`
RestructuringPolicy:
`Brent_restructuring`

ptcmbm : defmp

ErrorDistributionPolicy:
`Path_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`
BrentSplitCondition:
`Split_tree_weight`
RestructuringPolicy:
`Brent_restructuring`

dtcmbm : defmp

ErrorDistributionPolicy:
`Depth_weight_distribution`
ErrorRepresentationPolicy:
`Balanced_error_representation`
BrentSplitCondition:
`Split_tree_weight`
RestructuringPolicy:
`Brent_restructuring`

Figure 5.17: Configurations for both single- and multithreaded number types with Weighted Brent Restructuring and error bound balancing. For error bound balancing either the path weight error distribution (`ptcmb`, `ptcmbm`) or the depth weight error distribution (`dtcmb`, `dtcmbm`) is used.

medium amount of blocking nodes. As is, the depth heuristic does not handle the common subexpressions that are introduced by the restructuring algorithm correctly. If the paths between a common subexpression and the lowest common ancestor of its parents differ in length, the heuristic falsely decides that the smaller paths do not contribute to the parallel cost and strongly increases their required accuracies and therefore the requested accuracy at the common subexpression. Consequently, it increases not only the absolute, but also the parallel cost of the evaluation. The speedups for restructuring with and without error balancing methods are depicted in Figure 5.19. We observe that, as for the balanced tree, the speedup for the standard error distribution is higher than both balancing speedups (cf. Figure 5.16). In contrast to the previous results, the speedup for the depth weight error distribution is consistently higher than the speedup for the path weight error distribution. While this may in part be attributed to higher overall errors for the leaf nodes, the difference does not fully disappear for small numbers of blocking nodes where both error bound balancing methods lead to similar running times.

The definition of the depth weight error distribution leaves room for improvement. The impact of common subexpressions on the cost of the critical path is not handled properly and may even lead to an increase in parallel evaluation cost compared to the path weight error distribution. Furthermore, paths of the same length are currently treated to be

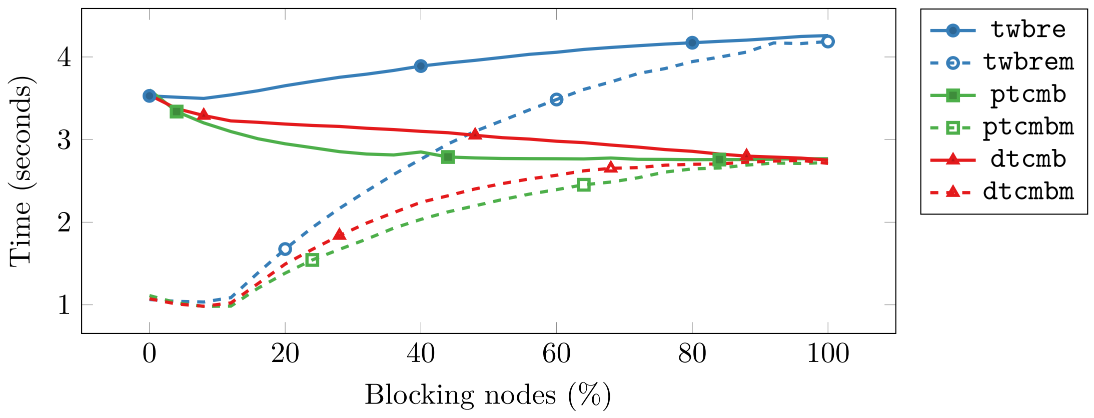


Figure 5.18: Parallel evaluation times for combined strategies on a list-like expression dag with 20000 operator nodes, random operators, target accuracy $z = -50000$ and various percentages of blocking nodes. For restructuring without error bound balancing, the running time increases fast as soon as blocking nodes are present. If error bound balancing is added to the restructuring, the effect is weakened. The path weight generally leads to a better running time and, in consequence, dominates the depth weight for the parallel evaluation as well.

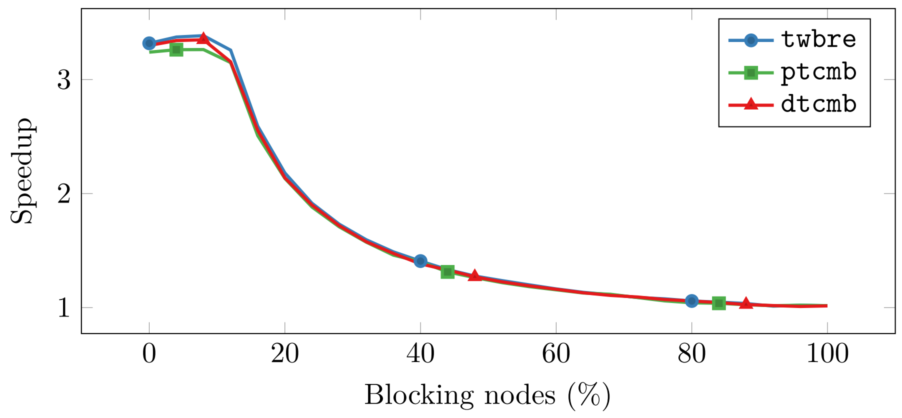


Figure 5.19: Speedup for combined strategies regarding the parallel evaluation of a list-like expression dag with 20000 operator nodes, random operators, target accuracy $z = -50000$ and various percentages of blocking nodes. The speedup for all strategies is similar. Introducing error bound balancing reduces the speedup slightly for a small number of blocking nodes.

equally likely to be the critical path, although their actual cost may differ widely due to operation constants and the size of their siblings. Nevertheless, further improvements are unlikely to make the depth heuristic competitive to the path weight unless the number of available processors increases significantly.

6 Error Bounds and Floating-Point Numbers

For the definition of error bound balancing and restructuring methods, we assumed that the error bounds and weights that occur in the respective algorithms are adequately representable in an implementation. In this chapter, we elaborate on the details regarding the error and weight representations and address some of the practical problems that arise in an implementation. In Section 6.1, we compare different methods for the representation of error bounds in an exact-decisions number type and thereby describe the error representation strategies that are provided for the configurable node type introduced in Chapter 3. In Section 6.2, we describe rounding issues that occur when a logarithmic representation based on floating-point primitives is used for very large or very small numbers. We then propose strategies for the correct rounding of the relevant arithmetic operations.

6.1 Error Bound Representation

Managing error bounds is a central part of any exact number type that is based on accuracy-driven evaluation. Typically, aside from a potential preceding floating-point filter, there are three situations in which error bound management takes place. First, approximations computed during the evaluation are stored together with error bounds in order to form intervals containing the actual values of the subexpressions. Second, in order to compute the operation constants for accuracy-driven evaluation, initial upper and lower bounds for each subexpression value must be present (cf. Section 4.1.1). Whenever possible, these initial bounds are acquired directly from a floating-point filter based on primitives. Otherwise, if the filter does not produce meaningful bounds, a computation at a higher precision takes place. Finally, during the main part of the evaluation, the required and requested accuracies for each node are computed and propagated to its subexpressions. Note that this algorithm might also be used as a subroutine during fixed-precision computation if invalid operations, such as a division by zero, must be ruled out. In summary, the error bound management includes

1. Error bound storage
2. Error bound computation during fixed-precision computation
3. Error bound propagation during accuracy-driven computation

For both the storage and the fixed-precision computation, bigfloat types are used in `leda::real` and the default version of `Real_algebraic` (not to be confused with the default configuration we used in the previous sections). The error bound propagation on the other hand is carried out with an integer exponent representation, using only the, appropriately rounded, logarithm of an error bound for the necessary computations. In each of the three cases a midpoint-radius representation of the error interval is used. The number type `Core::Expr` takes a similar approach for the error propagation, but uses logarithmic upper and lower bounds for an initial fixed-precision estimate. It stores both representations at each operator node.

6.1.1 Conversion Between Representations

Using different types of error representations for different parts of the evaluation requires constantly converting the representations to each other. In the standard version of `Real_algebraic`, the error bounds established during an accuracy-driven evaluation must be converted to bigfloat types when stored in the node. Conversely, established error bounds must be set into a logarithmic format in order to determine whether a requested accuracy is already guaranteed by a previously computed approximation. These conversions may be costly or have costly side effects. Converting an integer exponent representation to a bigfloat can be done exactly in a straightforward manner. In the opposite direction, creating an integer exponent representation from a bigfloat involves design choices that potentially have a negative impact on the running time.

```
mpfr_exp_t ceil_log2(const mpfr_t& a)
{
  return mpfr_get_exp(a);
}
```

(a) Inexact conversion

```
mpfr_exp_t ceil_log2(const mpfr_t& a)
{
  mpfr_exp_t e = mpfr_get_exp(a);
  mpfr_t rop; mpfr_init(rop);
  mpfr_div_2si(rop,a,e,MPFR_RNDA);
  if (mpfr_cmp_d(rop,0.5) == 0) --e;
  mpfr_clear(rop);
  return e;
}
```

(b) Exact conversion

Figure 6.1: Implementations for the `ceil_log2` function converting a bigfloat error representation into an integer exponent representation when using a `mpfr_t` bigfloat. Checking whether the stored number is a power of two involves an extraction of the mantissa and therefore a copy of the mantissa to a new bigfloat.

In general, a conversion from a bigfloat type to an integer exponent cannot be done exactly. We want the result of the conversion to represent an upper bound to the value stored in the bigfloat in order to never underestimate the actual error. Since bigfloats are usually stored internally by a mantissa m and an exponent e with $m \leq 1$, it is tempting to just use e as the value for the integer exponent representation. This conversion method is used in both `leda::real` and `Real_algebraic`. For $m \leq 1$ it always leads to a correct upper bound and for $m \in (0.5, 1]$ it leads to the closest possible correct integer value.

Due to the nature of floating-point representations, bigfloat number types such as `mpfr_t` and `leda::bigfloat` (if normalized) guarantee $m \in [0.5, 1)$ instead. The conversion method therefore overestimates the best possible conversion by one if the stored value is a power of two. One consequence of the described behavior is that the conversion from an integer exponent to a bigfloat and back to an integer exponent is never error-free, although in theory this should always be the case. This can have dramatic consequences on the running time if a (sub-)expression must be re-evaluated to the same accuracy. Due to the conversion, it cannot be detected that the computed result is already sufficient and the approximations in the whole subexpression are recomputed. When a recursive evaluation strategy is used, this may happen multiple times during a single evaluation of the expression dag. The problem occurs as well if the same decision is requested more than once by the user, which in theory should be a fast operation.

exact conversion

We call a conversion *exact* if it returns the closest possible upper bound for the converted value in the new format. Correcting for the conversion error is generally possible, but inefficient if conversions happen frequently. Checking whether the value of a bigfloat is a power of two is expensive. In the case of `MPFR`, it changes a simple read instruction on a member variable to a more elaborate method. Figure 6.1 shows the implementation of the method for the `Mpfr_approximation_policy` class in `Real_algebraic`. The exact implementation must access the mantissa in order to compare it with 0.5. For this, a new temporary bigfloat is created. We demonstrate the impact of this change experimentally. Figure 6.2 shows a configuration using the default error representation of `Real_algebraic`, i.e., a bigfloat representation for both storage and fixed-precision computation. Note that the default configuration for the previous experiments, as defined in Figure 4.30 (page 110), does not use the original error representation policy of `Real_algebraic`. Changing the behavior of the conversion method requires various modifications throughout the number type depending on the used approximation policy. Creating an explicit policy to support such a change is unnecessarily complicated, therefore we perform it manually. We call the configuration resulting from `bfl` by replacing each conversion by an exact variant `bflx`.

bfl : def

ErrorRepresentationPolicy:
`Default_error_representation`

Figure 6.2: Configuration for the default number type with a bigfloat error representation for storage and fixed-precision computation.

We compare the exact and the inexact implementation of the conversion on a randomly generated expression tree with random operators and exponentially ($\lambda = 1$) distributed `double` values as operands. Figure 6.3 shows the time needed for evaluating such trees to accuracy $z = -1000$. We can observe that changing the conversion function has a large impact on the overall running time. The loss of performance introduced by treating the above-mentioned special case seems to be disproportionally high. Similar experiments show that, especially for small expression dags, this difference can become even larger, doubling the running time needed for the evaluation [Wil18b]. Changing the conversion function in the proposed way is therefore not advisable. The exact conversion can be

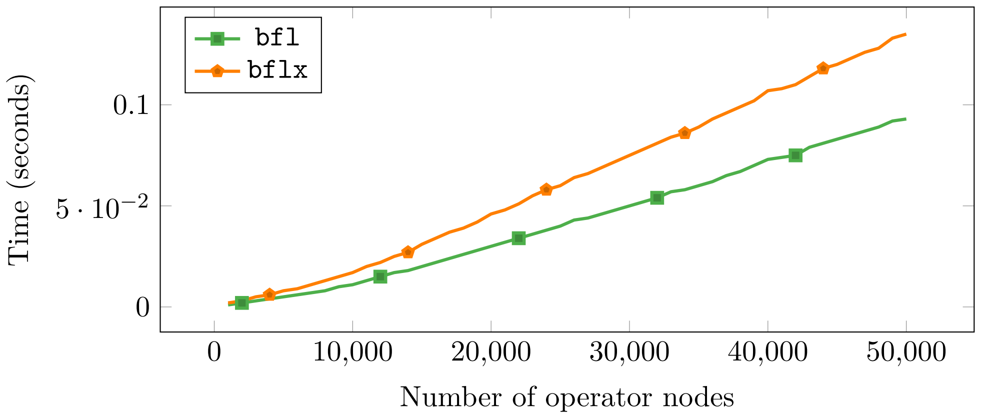


Figure 6.3: Evaluation times for a random expression tree with random operators to accuracy $z = -1000$, using the bigfloat error configuration with (`bflx`) and without (`bfl`) exact error conversion. If the representations are converted exactly, the running time increases by 40–50 %.

made more efficient by implementing it directly in the bigfloat number type. Alternatively, conversions can be made superfluous by switching to another error representation.

6.1.2 Exponent Representations for Fixed-Precision Computation

Error bound conversions can be avoided completely by using the same representation at each stage of the evaluation. Using a bigfloat representation for the error propagation would be very expensive and therefore does not seem to be a reasonable approach. Instead, integer exponents can be used for storing the error bound and, consequentially, for the fixed-precision computation. Storing and processing primitive integers is much faster than storing and processing bigfloats. During an accuracy-driven evaluation we do not lose any accuracy by using this representation since all occurring error bounds are represented as exponent anyway. Yet, the error bounds achieved by the initial fixed-precision computation can become much worse. While not too different for multiplications, divisions or roots, the increase in the error bound can be exponentially higher than with bigfloat representation for a series of additions.

The impact that a change in the error representation for the initial fixed-precision computation has on exact computation is shown exemplarily in Figure 6.4 for the computation of the Fibonacci numbers (cf. Figure 4.54, page 126). The running time for both configurations jumps whenever an additional iteration must be started in order to match the separation bound. The default configuration, which utilizes a pure exponent-based error representation, leads to a considerably lower initial accuracy than the bigfloat representation. However, the initial error bound is far less important than the final target accuracy. The target accuracy for an iteration is computed as the sum of the initial (absolute) accuracy and a steadily increasing relative accuracy. Since the relative accuracy is doubled during each iteration, one of the configurations might hit an accuracy close to the separation bound, whereas the other one closely misses the bound and starts

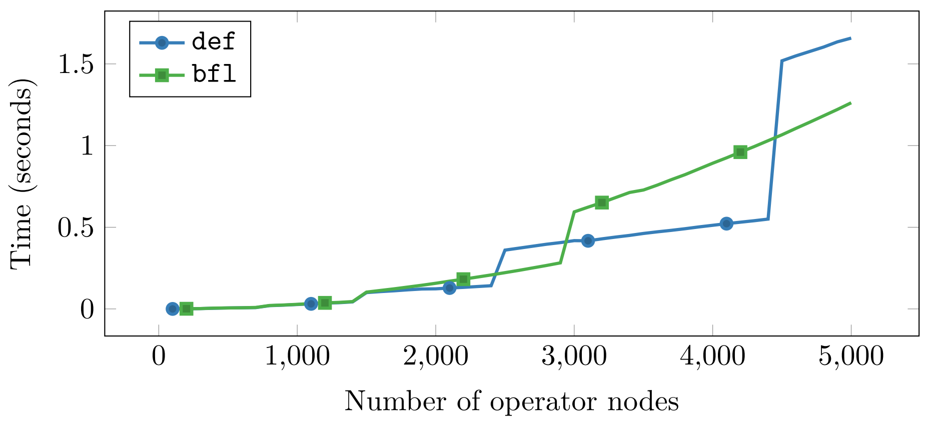


Figure 6.4: Evaluation times of the Fibonacci test for different error representations. With a bigfloat representation of the error, the initial accuracy is higher. Therefore, jumps caused by an additional iteration of the floating-point filter happen later.

a computation at almost twice the necessary accuracy. Hence, the configuration using exponents might still turn out to be faster than the configuration using bigfloats. This effect can be weakened by setting the target accuracy directly to the magnitude of the separation bound when it is sufficiently small or by defining a fixed number of iterations until the separation bound must be reached. Nevertheless, for each of those strategies there are cases in which a worse initial accuracy leads to a better running time. While this does not imply that the initial accuracy is irrelevant, the example demonstrates that chance plays a much larger role for the final running time than the initial error bound as long as the difference in magnitude of the initial error bounds is of the size of the final requested accuracy, i.e., the size of the magnitude of the separation bound if the result is zero. If the final accuracy needed for the evaluation is low, the reduction in the number of bigfloat operations caused by the change of the error representation usually compensates for the increase in the initial error bound.

6.1.3 Floating-Point Exponents

A compromise between the representations based on integer exponents and bigfloats can be found by employing floating-point primitives for an exponent representation. Operations on floating-point primitives such as `double` are generally much faster than on bigfloats. At the same time, they offer a more nuanced expression of the error bound than an integer exponent. With an integer representation, every increase in the error bound results in an increase by at least one order of magnitude. When using a `double`, the increase is possibly more fine-grained, although it eventually hits a minimum increase cap as well. A configuration for a number type using double exponent representations is depicted in Figure 6.5. Figure 6.6 shows the initial accuracy obtained by the fixed-precision computation for various error representation strategies. Up until about 1500 operator nodes, the error bound can be extracted from the filter policy. As soon as the

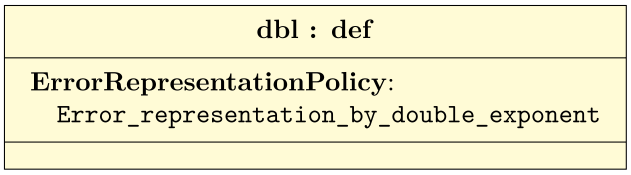

dbl : def
ErrorRepresentationPolicy:
Error_representation_by_double_exponent

Figure 6.5: Configuration based on `def` with a floating-point exponent error representation.

error bound does not fit into a `double` anymore, a fixed-precision computation is started and the three strategies lead to different initial accuracies. Each strategy shows a linear increase in magnitude, i.e., an exponential increase of the initial error bound. Note that all accuracy values in this case are positive and therefore all error bounds are larger than one. The error representation policy relying on floating-point exponents ranges in between the other two representations.

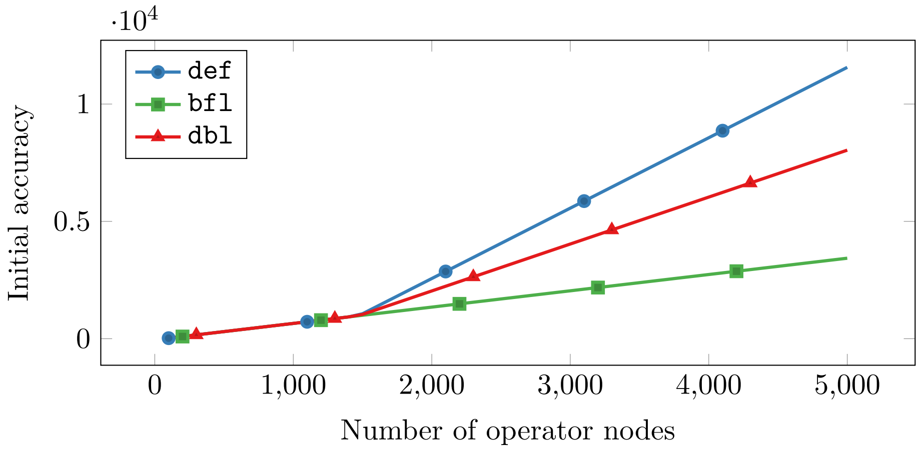


Figure 6.6: Initial accuracies determined by the fixed-precision computation during the evaluation of the Fibonacci test for different error representations. For small numbers of operands, the initial approximations are acquired from the floating-point filter and therefore are identical for all error representations. For 5000 operands, the bigfloat, double and integer representations lead to initial error bounds of 2^{3430}, 2^{8035} and 2^{11564}, respectively.

While having useful properties for the initial fixed-precision computation, the usage of floating-point exponents for error bound representations is especially relevant during the error propagation of an accuracy-driven evaluation since it enables error bound balancing as introduced in Section 4.2. As for the other representations, special measures must be taken in order to ensure correct rounding when handling these error bounds. In contrast to the other representations, doing so with minimal overhead is rather complicated. For the experiments performed in the previous chapters we mostly did not ensure correct rounding for double exponents in the interest of not distorting the results. While, in all likelihood, the operations where incorrect rounding may occur will never cause an incorrect decision, they should be adjusted before used in practice. In the next section, we have a closer look on how correct rounding can be achieved for the relevant use cases.

6.2 Rounding Floating-Point Operations

Error bounds used in an exact number type must always be guaranteed to be an upper bound to the actual error. When error bounds are represented as floating-point exponents, each operation on the error bound must be guaranteed to maintain this property. In particular, various arithmetic operations on floating-point numbers must be guaranteed to be rounded in the correct direction. For a correct handling of error bounds, all basic arithmetic operations must be available for the represented value, both rounded toward and away from infinity. This includes addition and subtraction, multiplication and division as well as p-th powers and d-th roots. If the error bound values are represented by their exponent, i.e., by their logarithm, multiplication and division translate to addition and subtraction and powers and roots translate to multiplication and division. Performing an addition or an subtraction with a representation based on the exponent requires more sophisticated methods since it involves computing base-2 powers and logarithms.

6.2.1 Basic Arithmetic Operations

All basic operations on floating-point primitives are guaranteed to be rounded correctly if the target processor and the compiler follow the IEEE 754 standard [IE754]. The standard defines four different rounding modes:

1. **Round to nearest:**
   Always round to the nearest representable floating-point number, with different strategies for tie-breaking.

2. **Round toward zero:**
   Always round to the value with smaller size.

3. **Round toward infinity:**
   Always round to the larger value.

4. **Round toward negative infinity:**
   Always round to the smaller value.

The first rounding mode, round to nearest, is the default rounding mode for g++. Unfortunately, this rounding mode is largely useless if an upper bound for the result is needed. Manually replacing the result by the next largest representable floating-point number is costly and unnecessarily introduces errors if the original result would have been exact. Hence, switching the rounding mode is crucial. A change of the rounding mode is expensive and should be used sparingly. Having a single change affect a large number of operations without simultaneously interfering with other parts of the evaluation requires carefully designed code. Most of the time rounding toward infinity is needed. For operations that must be rounded away from infinity, it can be avoided to switch the rounding mode by rearranging the respective arithmetic expressions. For example, if the term $a + b$ needs to be rounded down, the expression $-(-a - b)$ can be computed instead.

	To nearest	Nextafter	Always Up	Once Up	Restructured
Time (ms)	0.09	0.555	1.305	0.089	0.148

Table 6.7: Running times of 10^5 subsequent additions with various rounding methods. *Always Up* signifies that the rounding mode is changed to round toward infinity before each addition. *Once up* signifies that the rounding mode is changed only once.

Table 6.7 shows the running times for additions with different rounding methods. Manually jumping to the next largest floating-point number as well as switching the rounding mode for every operation is too expensive to be used in an implementation. If the new rounding mode is once set, however, the running time of the operations is similar to the running time of the operations with the default rounding mode enabled. Restructuring the expression to simulate rounding down while rounding toward infinity is enabled costs less than twice as much as the original operation. If used with caution, this technique can help to reduce the induced overhead.

6.2.2 Logarithm and Power

Error bounds must be added frequently during the evaluation, both for the initial fixed-precision computation and for the accuracy propagation phase. When two error bounds are represented by their exponents a and b, this equates to computing the value of

$$\mathrm{add}_{\mathrm{l}}(a, b) = \log(2^a + 2^b)$$

Being able to perform this operation with appropriate rounding is not only important for error bound management. If error bound balancing or weighted restructuring is used, weights are often represented logarithmically and computing the weight of a node usually requires adding the weights of its subexpressions. In this case, the values of 2^a and 2^b may not be representable by a primitive. An upper bound to the logarithm can be computed through repeated squaring [ML73]. If the difference between a and b is large, however, this process becomes numerically unstable. We pursue a different approach. Without loss of generality, assume that $a \geq b$. Then $\mathrm{add}_{\mathrm{l}}(a, b) = a + \log(1 + x)$ with $x = 2^{b-a} \leq 1$ and $x > 0$. Since the slope of the logarithm is monotonically decreasing, we can bound the second term by

$$\log(1 + x) \leq \log(1) + x \frac{d}{dz} \log(1 + z) \Big|_{z=0} = \frac{x}{\ln 2} \tag{6.1}$$

The factor $\frac{1}{\ln 2} \approx 1.44$ can be represented as an appropriately rounded constant. The value of x can be computed by a small product of values from a lookup table by considering the floating-point representation of $b - a$. Let $b - a = -m \cdot 2^e$ with $m \in [1, 2)$. Assuming `double` values, let $k = 52$, then the mantissa can be expressed as $m = \sum_{i=0}^{k} \beta_i 2^{-i}$ with $\beta_0 = 1$ and $\beta_i \in \{0, 1\}$ for all $1 \leq i \leq k$. We get

$$x = 2^{b-a} = 2^{(-\sum_{i=0}^{k} \beta_i 2^{e-i})} = \prod_{i=0}^{k} \left(2^{-2^{e-i}}\right)^{\beta_i}$$

The range of values taken by $e - i$ can be bounded to get a finite number of entries for a lookup table. If $e < -1$, then $b - a > -0.5$ and therefore $x > \frac{1}{\sqrt{2}} > \ln 2$. In this case the approximation in (6.1) would produce a value larger than one, although we know that $\log(1 + x) \leq 1$. Therefore, we can set $\mathrm{add}_\mathrm{l}(a, b) = a + 1$ without any further computations.

Conversely, if $e \geq \log(k + 1)$, then $b - a \leq -(k + 1)$ and $\frac{x}{\ln 2} < 2^{-k}$. If $|a| \geq 1$, then the sum $a + \frac{x}{\ln 2}$ is smaller than the next largest representable floating-point number from a and we can set $\mathrm{add}_\mathrm{l}(a, b) = \mathrm{nextafter}(a)$. Now assume $|a| < 1$. If a represents an error bound, the error bound is close to 1. In this case we just set $\mathrm{add}_\mathrm{l}(a, b) = a + 1 \geq 1$ without further computation. Having a slightly worse error bound for one special value does not notably affect the overall computation. For the tree weight or the tree operand weight, a logarithmic representation is used exclusively for large weights, hence $|a| \gg 1$. If a logarithmic representation is used for the path weight, we may scale up the weight function, such that each node has a weight of at least 2 (instead of 1) and therefore $|a| \geq 1$ for all operations.

7 Conclusion

Exact computation on large expressions gets slow for two reasons:

1. The cost for the evaluation of an operator node increases with the depth of the node in the expression dag.

2. The separation bound shrinks with the size of the expression and therefore the maximum target accuracy for an evaluation increases.

The cost increase along the edges of an expression dag dominates the evaluation cost if the value of the expression is non-zero, but close to zero. In Chapter 4, we have shown how the cost increase can be effectively reduced if the expression dag is unbalanced. Error bound balancing is a very reliable method to decrease the variable cost. It is nearly unaffected by the actual structure of the expression dag. Neither the choice of operators nor the presence of common subexpressions leads to unexpected behavior. The overhead caused by the computation of the weight is usually compensated by the increase in performance, except for small and already balanced expression dags. The path weight error distribution has shown to be generally superior to the proposed heuristics. Despite its more complicated definition, it was shown to admit a reasonably easy and efficient computation. All error bound balancing methods require a change in the error representation as well as the implementation of special measures to deal with rounding errors in the underlying floating-point primitives. In Chapter 6, it was shown that this can be realized without a significant loss in efficiency.

Restructuring bears an even larger potential than error bound balancing in reducing the cost of an evaluation, but it also bears a higher risk of worsening the performance. If applicable, restructuring almost completely eliminates the structure-related cost increase by not only reducing the variable cost but at the same time reducing the cost induced by the operator constants. It significantly increases parallelizability and there is reason to believe that a well-chosen restructuring method leads to a better ordering of the operator nodes, which, in turn, increases the chance on bigfloat conversions. However, the consequences of reordering are hard to predict and might as well have a negative impact on the evaluation. Blocking nodes reduce the benefits of restructuring. In general, Brent Restructuring paired with the tree weight heuristic is the method of choice. While AM-Balancing is reasonably safe to use, its use cases are too specific for a general purpose number type and it should be seen more as a proof of concept on the future development of non- or semi-invasive restructuring strategies. Weighted Brent Restructuring with a reasonable heuristic behaves identically to unit weight Brent Restructuring if the expression dag is a tree and proves to be superior if blocking nodes are present.

The cost increase due to the structure becomes less important if the value of the expression is zero. In this case, the expression must be evaluated at the maximum target accuracy as defined by the separation bound. In Chapter 5, we have shown that, for a high target accuracy and a sufficiently balanced expression dag, multithreading significantly reduces the evaluation time. A reduction in evaluation time takes place even for fairly unbalanced structures. In extremely unbalanced structures, executing a restructuring algorithm before the execution considerably increases the parallelizability of the expression. Furthermore, a depth-prioritization strategy for the task assignment in the thread pool produces a small but consistent advantage compared to a first-in, first-out task assignment.

In summary, path weight error bound balancing can safely be implemented as the default in a general purpose number type if paired with adequate, empirically determined bounds for a minimum number of nodes and a minimum deviation from the optimal depth. Likewise, depth-prioritized multithreading paired with bounds for a minimum number of nodes and a minimum target accuracy is a sensible addition to a number type. Weighted Brent Restructuring should be considered for large expression dags if the constructed graph is expected to be a tree or at least tree-like. In an adequate setting, it can greatly improve the running time. For a default implementation in a general purpose number type, further research on the relevant conditions is necessary. If a high number of processors is available, restructuring is a valuable tool to increase the parallelizability and, hence, the overall performance of the evaluation.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. “Models of Computation”. In: *The Design and Analysis of Computer Algorithms.* First edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1974, pp. 2–41.

[Alt79] Helmut Alt. “Square rooting is as difficult as multiplication”. In: *Computing* 21.3 (1979), pp. 221–232. DOI: `10.1007/BF02253055`.

[Bar86] Paul Barrett. “Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor”. In: *Advances in Cryptology - CRYPTO ’86, Santa Barbara, California, USA, 1986, Proceedings.* 1986, pp. 311–323. DOI: `10.1007/3-540-47721-7_24`.

[BBP01] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. “Interval arithmetic yields efficient dynamic filters for computational geometry”. In: *Discrete Applied Mathematics* 109.1-2 (2001), pp. 25–47. DOI: `10.1016/S0166-218X(00)00231-6`.

[BC90] Hans Boehm and Robert Cartwright. “Research Topics in Functional Programming”. In: ed. by David A. Turner. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. Chap. Exact Real Arithmetic Formulating Real Numbers As Functions, pp. 43–64. ISBN: 0-201-17236-4.

[BD97] David Berthelot and Marc Daumas. “Computing on sequences of embedded intervals”. In: *Reliable Computing* 3.3 (1997), pp. 219–227.

[Bel19] Fabrice Bellard. *The LibBF library.* Feb. 10, 2019. URL: `https://bellard.org/libbf/`.

[Ben+93a] Mohand O. Benouamer, Philippe Jaillon, Dominique Michelucci, and Jean-Michel Moreau. “A lazy exact arithmetic”. In: *11th Symposium on Computer Arithmetic, 29 June - 2 July 1993, Windsor, Canada, Proceedings.* Aug. 1993, pp. 242–249. DOI: `10.1109/ARITH.1993.378086`.

[Ben+93b] Mohand O. Benouamer, Philippe Jaillon, Dominique Michelucci, and Jean-Michel Moreau. “A lazy solution to imprecision in computational geometry”. In: *Proceedings of the 5th Canadian Conference on Computational Geometry, Waterloo, Ontario, Canada, August 1993.* 1993, pp. 73–78.

[BES02] Andrej Bauer, Martín Hötzel Escardó, and Alex K. Simpson. "Comparing Functional Paradigms for Exact Real-Number Computation". In: *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings.* 2002, pp. 488–500. DOI: 10.1007/3-540-45465-9_42.

[Bla02] Jens Blanck. *General purpose exact real arithmetic.* Tech. rep. CSR 21-200. 2002.

[BMS96] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. *The LEDA class real number.* Report MPI-I-1996-1-001. Saarbrücken, Germany: Max-Planck-Institut für Informatik, 1996.

[Bod+03] Gábor Bodnár, Barbara Kaltenbacher, Petru Pau, and Josef Schicho. "Exact Real Computation in Computer Algebra". In: *Symbolic and Numerical Scientific Computation.* Ed. by Franz Winkler and Ulrich Langer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 279–292. ISBN: 978-3-540-45084-9. DOI: 10.1007/3-540-45084-X_14.

[Bor16] Michele Borassi. "A note on the complexity of computing the number of reachable vertices in a digraph". In: *Information Processing Letters* 116.10 (2016), pp. 628–630. DOI: 10.1016/j.ipl.2016.05.002.

[Bre74] Richard P. Brent. "The Parallel Evaluation of General Arithmetic Expressions". In: *Journal of the ACM* 21.2 (1974), pp. 201–206. DOI: 10.1145/321812.321815.

[Bri06] Keith Briggs. "Implementing exact real arithmetic in python, C++ and C". In: *Theoretical Computer Science* 351.1 (2006), pp. 74–81. DOI: 10.1016/j.tcs.2005.09.058.

[Bur+00] Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn, and Stefan Schirra. "A Strong and Easily Computable Separation Bound for Arithmetic Expressions Involving Radicals". In: *Algorithmica* 27.1 (2000), pp. 87–99. DOI: 10.1007/s004530010005.

[Bur+04] Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. *The LEDA class real number – extended version.* Tech. rep. ECG-TR-363110-01. Max-Planck Institut für Informatik, Saarbrücken, Germany, 2004. URL: https://people.mpi-inf.mpg.de/~sschmitt/EXT.html.

[Bur+09] Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. "A Separation Bound for Real Algebraic Expressions". In: *Algorithmica* 55.1 (2009), pp. 14–28. DOI: 10.1007/s00453-007-9132-4.

[Bur+95] Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. "Exact geometric computation in LEDA". In: *Proceedings of the eleventh annual symposium on Computational geometry.* ACM. 1995, pp. 418–419.

[BVW08] Guy E. Blelloch, Virginia Vassilevska, and Ryan Williams. "A New Combinatorial Approach for Sparse Graph Problems". In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games.* 2008, pp. 108–120. DOI: `10.1007/978-3-540-70575-8_10`.

[BZ98] Christoph Burnikel and Joachim Ziegler. *Fast Recursive Division.* Report MPI-I-98-1-022. Saarbrücken, Germany: Max-Planck-Institut für Informatik, 1998.

[CA69] Stephen A. Cook and Stål O. Aanderaa. "On the minimum computation time of functions". In: *Transactions of the American Mathematical Society* 142 (1969), pp. 291–314. DOI: `10.2307/1995359`.

[Can88] John F. Canny. "Some Algebraic and Geometric Computations in PSPACE". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA.* 1988, pp. 460–467. DOI: `10.1145/62212.62257`.

[CGAL19] The CGAL Project. *Number type documentation. CGAL.* Version 4.14. Mar. 29, 2019. URL: `https://doc.cgal.org/4.14/Number_types/`.

[Che+01] Wei Chen, Xiaowen Deng, Koichi Wada, and Kimio Kawaguchi. "Constructing a Strongly Convex Superhull of Points". In: *International Journal of Computational Geometry & Applications* 11.5 (2001), pp. 487–502. DOI: `10.1142/S0218195901000614`.

[Che74] David R. Cheriton. "An extension to on-line multiplication lower bound results". In: *ACM SIGACT News* 6.4 (1974), pp. 24–31. DOI: `10.1145/1008318.1008321`.

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American journal of mathematics* 58.2 (1936), pp. 345–363. DOI: `10.2307/2371045`.

[Coo66] Stephen A. Cook. "On the minimum computation time of functions". PhD thesis. Harvard University, 1966.

[CR73] Stephen A. Cook and Robert A. Reckhow. "Time Bounded Random Access Machines". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375. DOI: `10.1016/S0022-0000(73)80029-7`.

[CS18] Microsoft Corporation. *Documentation of the BigInteger struct. C#.* Version 7.3. May 7, 2018. URL: `https://docs.microsoft.com/de-de/dotnet/api/system.numerics.biginteger`.

[CWK02] Wei Chen, Koichi Wada, and Kimio Kawaguchi. "Robust algorithms for constructing strongly convex hulls in parallel". In: *Theoretical Computer Science* 289.1 (2002), pp. 277–295. DOI: `10.1016/S0304-3975(01)00274-2`.

[Dav19] David Eberly. *Documentation on the arithmetic classes. GeometricTools.* Version 3.28. Aug. 29, 2019. URL: `https://www.geometrictools.com/Source/Arithmetic.html`.

[Dek71] Theodorus J. Dekker. "A floating-point technique for extending the available precision". In: *Numerische Mathematik* 18.3 (1971), pp. 224–242. DOI: `10.1007/BF01397083`.

[Deu12] Peter Deuflhard. "A short history of Newton's method". In: *Documenta Mathematica, Optimization stories* (2012), pp. 25–30.

[DFI03] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. "Algorithm engineering, Algorithmics Column". In: *Bulletin of the EATCS* 79 (2003), pp. 48–63. DOI: `10.1142/9789812562494_0006`.

[DMM05] Marc Daumas, Guillaume Melquiond, and César A. Muñoz. "Guaranteed Proofs Using Interval Arithmetic". In: *17th IEEE Symposium on Computer Arithmetic (ARITH-17 2005), 27-29 June 2005, Cape Cod, MA, USA*. 2005, pp. 188–195. DOI: `10.1109/ARITH.2005.25`.

[DOY94] Thomas Dube, Kouchi Ouchi, and Chee-Keng Yap. "Real/Expr: A precision-driven expression package". In: *4th MSI Workshop on Computational Geometry. Mathematical Sciences Institute, Cornell University. Oct 14-15, 1994*. 1994.

[DP98] Olivier Devillers and Franco P. Preparata. "A Probabilistic Analysis of the Power of Arithmetic Filters". In: *Discrete & Computational Geometry* 20.4 (1998), pp. 523–547. DOI: `10.1007/PL00009400`.

[DY93] Thomas Dubé and Chee-Keng Yap. *A Basis for Implementing Exact Geometric Algorithms.* Extended Abstract. 1993.

[FB91] Shiaofen Fang and Beat D. Brüderlin. "Robustness in Geometric Modeling - Tolerance-Based Methods". In: *Computational Geometry - Methods, Algorithms and Applications, International Workshop on Computational Geometry CG'91, Bern, Switzerland, March 21-22, 1991.* 1991, pp. 85–101. DOI: `10.1007/3-540-54891-2_7`.

[Fly70] Michael J. Flynn. "On Division by Functional Iteration". In: *IEEE Transactions on Computers* 19.8 (1970), pp. 702–706. DOI: `10.1109/T-C.1970.223019`.

[FM91] Steven Fortune and Victor Milenkovic. "Numerical Stability of Algorithms for Line Arrangements". In: *Proceedings of the Seventh Annual Symposium on Computational Geometry, North Conway, NH, USA, June 10-12, 1991.* 1991, pp. 334–341. DOI: `10.1145/109648.109685`.

[FNS04] Andras Frankel, Doron Nussbaum, and Jörg-Rüdiger Sack. "Floating-Point Filter for the Line Intersection Algorithm". In: *Geographic Information Science, Third International Conference, GIScience 2004, Adelphi, MD, USA, October 20-23, 2004, Proceedings.* 2004, pp. 94–105. DOI: 10.1007/978-3-540-30231-5_7.

[For89] Steven Fortune. "Stable Maintenance of Point Set Triangulations in Two Dimensions". In: *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989.* 1989, pp. 494–499. DOI: 10.1109/SFCS.1989.63524.

[Fou+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A multiple-precision binary floating-point library with correct rounding". In: *ACM Transactions on Mathematical Software* 33.2 (2007), p. 13. DOI: 10.1145/1236463.1236468. URL: https://www.mpfr.org/.

[Für09] Martin Fürer. "Faster Integer Multiplication". In: *SIAM Journal on Computing* 39.3 (2009), pp. 979–1005. DOI: 10.1137/070711761.

[FW93a] Steven Fortune and Christopher J. Van Wyk. "Efficient Exact Arithmetic for Computational Geometry". In: *Proceedings of the Ninth Annual Symposium on Computational GeometrySan Diego, CA, USA, May 19-21, 1993.* 1993, pp. 163–172. DOI: 10.1145/160985.161015.

[FW93b] Michael L. Fredman and Dan E. Willard. "Surpassing the Information Theoretic Bound with Fusion Trees". In: *Journal of Computer and System Sciences* 47.3 (1993), pp. 424–436. DOI: 10.1016/0022-0000(93)90040-4.

[Gia99] Pietro Di Gianantonio. "An Abstract Data Type for Real Numbers". In: *Theoretical Computer Science* 221.1-2 (1999), pp. 295–326. DOI: 10.1016/S0304-3975(99)00036-5.

[GL00] Paul Gowland and David R. Lester. "A Survey of Exact Arithmetic Implementations". In: *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers.* 2000, pp. 30–47. DOI: 10.1007/3-540-45335-0_3.

[Gla+17] Brian Gladman, William Hart, Jason Moxham, and the MPIR development team. *MPIR: Multiple Precision Integers and Rationals.* Version 3.0.0. Mar. 1, 2017. URL: http://mpir.org/.

[GMP16] Torbjörn Granlund and the GMP development team. *GNU MP Multiple precision arithmetic library.* Version 6.1.2. Dec. 16, 2016. URL: http://gmplib.org/.

[GO04] "Applications of Discrete and Computational Geometry". In: *Handbook of Discrete and Computational Geometry.* Ed. by Jacob E. Goodman and Joseph O'Rourke. Second edition. New York: Chapman & Hall/CRC Press, 2004. Chap. 45–63.

[Go19] Go Development Team. *Documentation of the big package. Go.* Version 1.12.7. July 8, 2019. URL: `https://golang.org/pkg/math/big/`.

[GPG19] The GnuPG Project. *The Libgcrypt library on the website of the GNU Privacy Guard (GnuPG).* Version 1.8.5. Aug. 29, 2019. URL: `https://gnupg.org/software/libgcrypt/`.

[Grz55] Andrzej Grzegorczyk. “Computable functionals”. In: *Fundamenta Mathematicae* 42 (1955), pp. 168–202.

[GSS89] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. “Epsilon Geometry: Building Robust Algorithms from Imprecise Computations”. In: *Proceedings of the Fifth Annual Symposium on Computational Geometry, Saarbrücken, Germany, June 5-7, 1989.* 1989, pp. 208–217. DOI: `10.1145/73833.73857`.

[GW19] Hanna Geppert and Martin Wilhelm. “Internal versus external balancing in the evaluation of graph-based number types”. In: *Special Event on Analysis of Experimental Algorithms, SEA^2 2019, June 24-29, 2019, Kalamata, Greece.* 2019.

[Has97] Karl Hasselström. “Fast Division of Large Integers”. MA thesis. Stockholm, Sweden: KTH Royal Institute of Technology, 1997.

[HH19] David Harvey and Joris van der Hoeven. “Integer multiplication in time O(n log n)”. Preprint <hal-02070778>. Mar. 2019. URL: `https://hal.archives-ouvertes.fr/hal-02070778`.

[HHK89] Christoph M. Hoffmann, John E. Hopcroft, and Michael Karasick. “Robust set operations on polyhedral solids”. In: *IEEE Computer Graphics and Applications* 9.6 (1989), pp. 50–59. DOI: `10.1109/38.41469`.

[HK14] Bruno Haible and Richard B. Kreckel. *The Class Library for Numbers.* Version 1.3.4. Oct. 16, 2014. URL: `https://ginac.de/CLN/`.

[Hoe01] Joris van der Hoeven. *Zero-testing, witness conjectures and differential diophantine approximation.* Tech. rep. 2001-62. 2001.

[Hoe06a] Joris van der Hoeven. “Computations with effective real numbers”. In: *Theoretical Computer Science* 351.1 (2006), pp. 52–60. DOI: `10.1016/j.tcs.2005.09.060`.

[Hoe06b] Joris van der Hoeven. “Counterexamples to witness conjectures”. In: *Journal of Symbolic Computation* 41.9 (2006), pp. 959–963. DOI: `10.1016/j.jsc.2006.04.008`.

[Hoe06c] Joris van der Hoeven. “Effective real numbers in Mmxlib”. In: *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings.* 2006, pp. 138–145. DOI: `10.1145/1145768.1145795`.

[Hoe97] Joris van der Hoeven. “Asymptotique automatique”. PhD thesis. École polytechnique, France, 1997, pp. 177–179.

[Hof01] Christoph M. Hoffmann. “Robustness in Geometric Computations”. In: *Journal of Computing and Information Science in Engineering* 1.2 (2001), pp. 143–155. DOI: 10.1115/1.1375815.

[HS06] Joris van der Hoeven and John Shackell. “Complexity bounds for zero-test algorithms”. In: *Journal of Symbolic Computation* 41.9 (2006), pp. 1004–1020. DOI: 10.1016/j.jsc.2006.06.001.

[IE754] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[Jai93] Philippe Jaillon. “Proposition d’une arithmétique rationnelle paresseuse et d’un outil d’aide à la saisie d’objets en synthèse d’images. (Proposal for a Lazy Rational Arithmetic and a Tool for the Simplification of the input of Objects in Computer Graphics)”. PhD thesis. École nationale supérieure des mines de Saint-Étienne, France, 1993. URL: https://tel.archives-ouvertes.fr/tel-00822902.

[Jav19] Oracle Corporation. *Documentation of the class BigInteger. Java.* Version 12. Mar. 25, 2019. URL: https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html.

[JS19] Mozilla Foundation. *Documentation of the BigInt object in JavaScript.* Aug. 26, 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt.

[Kar+99] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee-Keng Yap. “A Core Library for Robust Numeric and Geometric Computation”. In: *Proceedings of the Fifteenth Annual Symposium on Computational Geometry, Miami Beach, Florida, USA, June 13-16, 1999.* 1999, pp. 351–359. DOI: 10.1145/304893.304989.

[Ket+08] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap. “Classroom examples of robustness problems in geometric computations”. In: *Computational Geometry* 40.1 (2008), pp. 61–78. DOI: 10.1016/j.comgeo.2007.06.003.

[KLN91] Michael S. Karasick, Derek Lieber, and Lee R. Nackman. “Efficient Delaunay Triangulation using Rational Arithmetic”. In: *ACM Trans. Graph.* 10.1 (1991), pp. 71–91. DOI: 10.1145/99902.99905.

[Knu92] Donald E. Knuth. “Parsimonious algorithms”. In: *Axioms and Hulls.* Vol. 606. Lecture Notes in Computer Science. Springer, 1992, pp. 61–67. ISBN: 3-540-55611-7. DOI: 10.1007/3-540-55611-7.

[KO62] Anatolii Alexeevich Karatsuba and Yuri Petrovich Ofman. “Multiplication of many-digital numbers by automatic computers”. In: *Doklady Akademii Nauk.* Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.

[Ko91] Ker-I Ko. *Complexity Theory of Real Functions.* Birkhäuser / Springer, 1991. ISBN: 978-1-4684-6802-1. DOI: 10.1007/978-1-4684-6802-1.

[KR06] Vladik Kreinovich and Siegfried M. Rump. “Towards Optimal Use of Multi-Precision Arithmetic: A Remark”. In: *Reliable Computing* 12.5 (2006), pp. 365–369. DOI: `10.1007/s11155-006-9007-4`.

[KS99] Simon Kahan and Jack Snoeyink. “On the bit complexity of minimum link paths: Superquadratic algorithms for problem solvable in linear time”. In: *Computational Geometry* 12.1-2 (1999), pp. 33–44. DOI: `10.1016/S0925-7721(98)00041-8`.

[Kus84] Boris Abramovich Kushner. *Lectures on constructive mathematical analysis.* Vol. 60. Providence, Rhode Island: American Mathematical Society, 1984.

[Lam07] Branimir Lambov. “RealLib: An efficient implementation of exact real arithmetic”. In: *Mathematical Structures in Computer Science* 17.1 (2007), pp. 81–98. DOI: `10.1017/S0960129506005822`.

[Lév16] Bruno Lévy. “Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK)”. In: *Computer-Aided Design* 72 (2016), pp. 3–12. DOI: `10.1016/j.cad.2015.10.004`.

[Liu+16] Yong-Jin Liu, Cheng-Chi Yu, Minjing Yu, Kai Tang, and Deok-Soo Kim. “A Robust Divide and Conquer Algorithm for Progressive Medial Axes of Planar Shapes”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.12 (2016), pp. 2522–2536. DOI: `10.1109/TVCG.2015.2511739`.

[LPY05] Chen Li, Sylvain Pion, and Chee-Keng Yap. “Recent progress in exact geometric computation”. In: *The Journal of Logic and Algebraic Programming* 64.1 (2005), pp. 85–111. DOI: `10.1016/j.jlap.2004.07.006`.

[LTM19] Team libtom. *The LibTomMath library.* Version 1.1.0. Jan. 28, 2019. URL: `https://www.libtom.net/LibTomMath/`.

[LY01] Chen Li and Chee-Keng Yap. “A new constructive root bound for algebraic expressions”. In: *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.* 2001, pp. 496–505.

[LY07] Yong Li and Jun-Hai Yong. “Efficient Exact Arithmetic over Constructive Reals”. In: *Theory and Applications of Models of Computation, 4th International Conference, TAMC 2007, Shanghai, China, May 22-25, 2007, Proceedings.* 2007, pp. 440–449. DOI: `10.1007/978-3-540-72504-6_40`.

[Mah62] Kurt Mahler. “On some inequalities for polynomials in several variables”. In: *Journal of the London Mathematical Society* 37.1 (1962), pp. 341–344.

[Maz63] Stanisław Mazur. *Computable analysis.* Vol. 33. Rozprawy Matematyczne. Warsaw, 1963.

[Mbed19] ARM Limited. *The mbed TLS library.* Version 2.16.2. June 11, 2019. URL: `https://tls.mbed.org/`.

[Mig82] Maurice Mignotte. “Identification of Algebraic Numbers”. In: *Journal of Algorithms* 3.3 (1982), pp. 197–204. DOI: `10.1016/0196-6774(82)90019-0`.

[Mil88] Victor Milenkovic. “Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic”. In: *Artificial Intelligence* 37.1-3 (1988), pp. 377–401. DOI: `10.1016/0004-3702(88)90061-6`.

[MK19] John Maddock and Christopher Kormanyos. *Boost.Multiprecision Library Documentation. Boost.* Version 1.71.0. Apr. 12, 2019. URL: `https://www.boost.org/doc/libs/1_71_0/libs/multiprecision/`.

[ML73] Jayanti C. Majithia and D. Levan. “A note on base-2 logarithm computations”. In: *Proceedings of the IEEE* 61.10 (1973), pp. 1519–1520. DOI: `10.1109/PROC.1973.9318`.

[MN89] Kurt Mehlhorn and Stefan Näher. “LEDA: A Library of Efficient Data Types and Algorithms”. In: *Mathematical Foundations of Computer Science 1989, MFCS’89, Porabka-Kozubnik, Poland, August 28 - September 1, 1989, Proceedings.* 1989, pp. 88–106. DOI: `10.1007/3-540-51486-4_58`.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999. ISBN: 0-521-56329-1. URL: `https://people.mpi-inf.mpg.de/~mehlhorn/LEDAbook.html`.

[Mnë88] Nikolai E. Mnëv. “The universality theorems on the classification problem of configuration varieties and convex polytopes varieties”. In: *Topology and geometry - Rohlin seminar.* Ed. by Oleg Yanovich Viro and Anatoly Moiseevich Vershik. Vol. 1346. Lecture Notes in Mathematics. Berlin, Heidelberg, 1988, pp. 527–543.

[Moo19] Paul Moore. *Boost.Rational Library Documentation. Boost.* Version 1.71.0. Apr. 12, 2019. URL: `https://www.boost.org/doc/libs/1_71_0/libs/rational/`.

[Mör10] Marc Mörig. “Deferring Dag Construction by Storing Sums of Floats Speeds-Up Exact Decision Computations Based on Expression Dags”. In: *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings.* 2010, pp. 109–120. DOI: `10.1007/978-3-642-15582-6_23`.

[Mör15a] Marc Mörig. “Algorithm Engineering for Expression Dag Based Number Types”. PhD thesis. Otto-von-Guericke Universität Magdeburg, 2015. URL: `http://dx.doi.org/10.25673/4246`.

[Mör15b] Marc Mörig. “Another Classroom Example of Robustness Problems in Planar Convex Hull Computation”. In: *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers.* 2015, pp. 446–450. DOI: `10.1007/978-3-319-32859-1_38`.

[MPB19] Guillaume Melquiond, Sylvain Pion, and Hervé Brönnimann. *Boost Interval Arithmetic Library Documentation. Boost.* Version 1.71.0. Apr. 12, 2019. URL: `https://www.boost.org/doc/libs/1_71_0/libs/numeric/interval/`.

[MRS10] Marc Mörig, Ivo Rössling, and Stefan Schirra. "On Design and Implementation of a Generic Number Type for Real Algebraic Number Computations Based on Expression Dags". In: *Mathematics in Computer Science* 4.4 (2010), pp. 539–556. DOI: `10.1007/s11786-011-0086-1`.

[MS07] Marc Mörig and Stefan Schirra. "On the Design and Performance of Reliable Geometric Predicates using Error-free Transformations and Exact Sign of Sum Algorithms". In: *Proceedings of the 19th Annual Canadian Conference on Computational Geometry, CCCG 2007, August 20-22, 2007, Carleton University, Ottawa, Canada.* 2007, pp. 45–48.

[MS10] Matthias Müller-Hannemann and Stefan Schirra, eds. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice.* Vol. 5971. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-14865-1. DOI: `10.1007/978-3-642-14866-8`.

[MS15] Marc Mörig and Stefan Schirra. "Precision-Driven Computation in the Evaluation of Expression-Dags with Common Subexpressions: Problems and Solutions". In: *6th International Conference on Mathematical Aspects of Computer and Information Sciences, MACIS.* 2015, pp. 451–465. DOI: `10.1007/978-3-319-32859-1_39`.

[Mül00] Norbert Th. Müller. "The iRRAM: Exact Arithmetic in C++". In: *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers.* 2000, pp. 222–252. DOI: `10.1007/3-540-45335-0_14`.

[OF97] Stuart F. Oberman and Michael J. Flynn. "Division Algorithms and Implementations". In: *IEEE Transactions on Computers* 46.8 (1997), pp. 833–854. DOI: `10.1109/12.609274`.

[OSSL19] OpenSSL Software Foundation. *The OpenSSL Toolkit.* Version 1.1.1. May 28, 2019. URL: `https://www.openssl.org/`.

[Ouc97] Kouji Ouchi. "Real/Expr: Implementation of an exact computation package". MA thesis. New York University, Department of Computer Science, Courant Institute, 1997.

[PF11] Sylvain Pion and Andreas Fabri. "A generic lazy evaluation scheme for exact geometric computations". In: *Science of Computer Programming* 76.4 (2011), pp. 307–323. DOI: `10.1016/j.scico.2010.09.003`.

[PFM74] Michael S. Paterson, Michael J. Fischer, and Albert R. Meyer. *An Improved Overlap Argument for On-Line Multiplication.* Tech. rep. Massachusetts Institute of Technology, 1974.

[PHP19] The PHP group. *BCMath Arbitrary Precision Mathematics. PHP.* Version 7.3.9. Aug. 29, 2019. URL: `https://www.php.net/manual/en/book.bc.php`.

[PR89] Marian Boykan Pour-El and Jonathan Ian Richards. *Computability in analysis and physics.* Perspectives in Mathematical Logic. Springer, 1989. ISBN: 978-3-540-50035-3.

[Prl19] Perl 5 Porters. *Documentation of the bigint pragma. Perl.* Version 5.30.0. May 22, 2019. URL: `https://perldoc.perl.org/bigint.html`.

[PY06] Sylvain Pion and Chee-Keng Yap. "Constructive root bound for k-ary rational input numbers". In: *Theoretical Computer Science* 369.1-3 (2006), pp. 361–376. DOI: `10.1016/j.tcs.2006.09.010`.

[Pyt19] Python Software Foundation. *Documentation of the numeric types in Python. Python.* Version 3.7.4. July 8, 2019. URL: `https://docs.python.org/3/library/stdtypes.html#typesnumeric`.

[QYZ19] Meng Qi, Ke Yan, and Yuanjie Zheng. "GPredicates: GPU Implementation of Robust and Adaptive Floating-Point Predicates for Computational Geometry". In: *IEEE Access* 7 (2019), pp. 60868–60876. DOI: `10.1109/ACCESS.2019.2911641`.

[RA15] Marc Mörig and Stefan Schirra. *RealAlgebraic - a number type for exact geometric computation.* Nov. 30, 2015. URL: `http://www.isg.cs.uni-magdeburg.de/ag/RealAlgebraic/`.

[RE03] Daniel Richardson and Ahmed El-Sonbaty. "Use of algebraically independent numbers for zero recognition of polynomial terms". In: *Journal of Complexity* 19.5 (2003), pp. 631–637. DOI: `10.1016/S0885-064X(03)00047-5`.

[RE06] Daniel Richardson and Ahmed El-Sonbaty. "Counterexamples to the uniformity conjecture". In: *Computational Geometry* 33.1-2 (2006), pp. 58–64. DOI: `10.1016/j.comgeo.2004.02.005`.

[Ric00] Daniel Richardson. "The Uniformity Conjecture". In: *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers.* 2000, pp. 253–272. DOI: `10.1007/3-540-45335-0_15`.

[Ric97] Daniel Richardson. "How to Recognize Zero". In: *Journal of Symbolic Computation* 24.6 (1997), pp. 627–645. DOI: `10.1006/jsco.1997.0157`.

[Ros39] John Barkley Rosser Sr. "An Informal Exposition of Proofs of Gödel's Theorems and Church's Theorem". In: *J. Symb. Log.* 4.2 (1939), pp. 53–60. DOI: `10.2307/2269059`.

[RR05] Nathalie Revol and Fabrice Rouillier. "Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library". In: *Reliable Computing* 11.4 (2005), pp. 275–290. DOI: `10.1007/s11155-005-6891-y`.

[RR99] Helmut Ratschek and Jon G. Rokne. "Exact computation of the sign of a finite sum". In: *Applied Mathematics and Computation* 99.2-3 (1999), pp. 99–127. DOI: `10.1016/S0096-3003(98)00010-1`.

[Rub19] Ruby Development Team. *Documentation of the Integer object. Ruby.* Version 2.6.3. Apr. 17, 2019. URL: `https://ruby-doc.org/core-2.6.4/Integer.html`.

[San09] Peter Sanders. "Algorithm Engineering - An Attempt at a Definition". In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday.* 2009, pp. 321–340. DOI: `10.1007/978-3-642-03456-5_22`.

[Sch00] Stefan Schirra. "Robustness and Precision Issues in Geometric Computation". In: *Handbook of Computational Geometry.* Elsevier, 2000, pp. 597–632.

[Sch04] Susanne Schmitt. *Improved separation bounds for the diamond operator.* Report ECG-TR-363108-01. Sophia Antipolis, France: Effective Computational Geometry for Curves and Surfaces, 2004.

[Sch05] Susanne Schmitt. "The Diamond Operator - Implementation of Exact Real Algebraic Numbers". In: *Computer Algebra in Scientific Computing, 8th International Workshop, CASC 2005, Kalamata, Greece, September 12-16, 2005, Proceedings.* 2005, pp. 355–366. DOI: `10.1007/11555964_30`.

[Sch09] Stefan Schirra. "Much Ado about Zero". In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday.* 2009, pp. 408–421. DOI: `10.1007/978-3-642-03456-5_27`.

[Sch91] Peter Schorn. "Robust algorithms in a program library for geometric computation". PhD thesis. Swiss Federal Institute of Technology Zürich, 1991. DOI: `10.3929/ethz-a-000604568`.

[Sek04] Hiroshi Sekigawa. "Zero determination of algebraic numbers using approximate computation and its application to algorithms in computer algebra". PhD thesis. PhD thesis, University of Tokyo, 2004.

[Sha78] Michael Ian Shamos. "Computational Geometry". PhD thesis. Yale University, 1978.

[She97] Jonathan Richard Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". In: *Discrete & Computational Geometry* 18.3 (1997), pp. 305–368. DOI: `10.1007/PL00009321`.

[SI89] Kokichi Sugihara and Masao Iri. "Two design principles of geometric algorithms in finite-precision arithmetic". In: *Applied Mathematics Letters* 2.2 (1989), pp. 203–206. DOI: `10.1016/0893-9659(89)90022-0`.

[SS09] Kiyoshi Shirayanagi and Hiroshi Sekigawa. "Reducing exact computations to obtain exact results based on stabilization techniques". In: *Symbolic Numeric Computation, SNC '09, Kyoto, Japan - August 03 - 05, 2009.* 2009, pp. 191–198. DOI: `10.1145/1577190.1577219`.

[SS71] Arnold Schönhage and Volker Strassen. "Schnelle Multiplikation großer Zahlen". In: *Computing* 7.3-4 (1971), pp. 281–292. DOI: `10.1007/BF02242355`.

[ST05] Håkan Sundell and Philippas Tsigas. “Fast and lock-free concurrent priority queues for multi-thread systems”. In: *Journal of Parallel and Distributed Computing* 65.5 (2005), pp. 609–627. DOI: `10.1016/j.jpdc.2004.12.005`.

[Ste73] Pat H Sterbenz. *Floating-point computation.* Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

[Sug+00] Kokichi Sugihara, Masao Iri, Hiroshi Inagaki, and Toshiyuki Imai. “Topology-Oriented Implementation - An Approach to Robust Geometric Algorithms”. In: *Algorithmica* 27.1 (2000), pp. 5–20. DOI: `10.1007/s004530010002`.

[SVH89] Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic.* Tech. rep. 2. 1989. URL: `https://sourceforge.net/projects/bigz/`.

[SW17] Stefan Schirra and Martin Wilhelm. “On Interval Methods with Zero Rewriting and Exact Geometric Computation”. In: *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings.* 2017, pp. 211–226. DOI: `10.1007/978-3-319-72453-9_15`.

[Too63] Andrei Leonovich Toom. “The complexity of a scheme of functional elements realizing the multiplication of integers”. In: *Soviet Mathematics Doklady.* Vol. 3. 4. 1963, pp. 714–716.

[Tur37] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265. DOI: `10.1112/plms/s2-42.1.230`.

[Uhr17] Christian Uhrig. *The LEDA User Manual.* Version 6.5. Apr. 7, 2017. URL: `http://www.algorithmic-solutions.info/leda_manual/MANUAL.html`.

[Wei00] Klaus Weihrauch. *Computable Analysis - An Introduction.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000. ISBN: 978-3-540-66817-6. DOI: `10.1007/978-3-642-56999-9`.

[Wil17] Martin Wilhelm. “Balancing Expression Dags for More Efficient Lazy Adaptive Evaluation”. In: *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings.* 2017, pp. 19–33. DOI: `10.1007/978-3-319-72453-9_2`.

[Wil18a] Martin Wilhelm. *Multithreading for the expression-dag-based number type Real_algebraic.* Tech. rep. FIN-001-2018. Otto-von-Guericke-Universität Magdeburg, 2018.

[Wil18b] Martin Wilhelm. “On error representation in exact-decisions number types”. In: *Proceedings of the 30th Canadian Conference on Computational Geometry, CCCG 2018, August 8-10, 2018, University of Manitoba, Winnipeg, Manitoba, Canada.* 2018, pp. 367–373.

[Wil18c] Martin Wilhelm. “Restructuring Expression Dags for Efficient Parallelization”. In: *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L’Aquila, Italy.* 2018, 20:1–20:13. DOI: 10.4230/LIPIcs.SEA.2018.20.

[Yap97] Chee-Keng Yap. “Towards Exact Geometric Computation”. In: *Computational Geometry* 7 (1997), pp. 3–23. DOI: 10.1016/0925-7721(95)00040-2.

[Yap98] Chee-Keng Yap. “A new number core for robust numerical and geometric libraries”. In: *3rd CGC Workshop on Geometric Computing, Brown University, Rhode Island, USA, Oct 11-12, 1998.* Invited Talk. 1998.

[YD95] Chee-Keng Yap and Thomas Dubé. “The exact computation paradigm”. In: *Computing in Euclidean Geometry.* World Scientific, 1995, pp. 452–492. DOI: 10.1142/9789812831699_0011.

[Yu+10] Jihun Yu, Chee-Keng Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. “The Design of Core 2: A Library for Exact Numeric Computation in Geometry and Algebra”. In: *Proceedings of the Third International Congress on Mathematical Software, ICMS.* 2010, pp. 121–141. DOI: 10.1007/978-3-642-15582-6_24.

[ZFB93] Xiaohong Zhu, Shiaofen Fang, and Beat D. Brüderlin. “Obtaining robust Boolean set operations for manifold solids by avoiding and eliminating redundancy”. In: *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications, Montreal, Canada, May 19-21, 1993.* 1993, pp. 147–154. DOI: 10.1145/164360.164413.

[ZXY16] Yinhe Zheng, Lu Xia, and Qingchun Yu. “Identifying rock blocks based on exact arithmetic”. In: *International Journal of Rock Mechanics and Mining Sciences* 86 (2016), pp. 80–90. DOI: 10.1016/j.ijrmms.2016.03.020.

Index