The LEDA class **real** number*

Christoph Burnikel   Kurt Mehlhorn   Stefan Schirra

January 16, 1996

Abstract

We describe the implementation of the LEDA [MN95, Näh95] data type **real**. Every integer is a real and reals are closed under the operations addition, subtraction, multiplication, division and squareroot. The main features of the data type real are

- The user–interface is similar to that of the built–in data type double.
- All comparison operators $\{>, \geq, <, \leq, =\}$ are *exact*. In order to determine the sign of a real number x the data type first computes a rational number q such that $|x| \leq q$ implies $x = 0$ and then computes an approximation of x of sufficient precision to decide the sign of x. The user may assist the data type by providing a separation bound q.
- The data type also allows to evaluate real expressions with arbitrary precision. One may either set the mantissae length of the underlying floating point system and then evaluate the expression with that mantissa length or one may specify an error bound q. The data type then computes an approximation with absolute error at most q.

The implementation of the data type real is based on the LEDA data types **integer** and **bigfloat** which are the types of arbitrary precision integers and floating point numbers, respectively. The implementation takes various shortcuts for increased efficiency, e.g., a **double** approximation of any real number together with an error bound is maintained and tests are first performed on these approximations. A high precision computation is only started when the test on the **double** approximation is inconclusive.

*This work was supported in part by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141(ALCOM II) and the BMFT(Förderungskennzeichen ITS 9103).

Contents

1. Introduction

The need for exact real arithmetic or at least higher precision arithmetic than standard **double** precision arithmetic arises frequently. One prime example is computational geometry. Other examples are computational metrology and constrained logic programming; we refer the reader to [YD95] for an extensive discussion. Almost all algorithms in computational geometry are formulated under the assumption of exact real arithmetic. When implemented naively using floating point arithmetic these algorithms tend to be unreliable, i.e., to either crash or produce wrong results at least on some inputs. Two approaches have been taken to remedy this situation. The first approach sticks with floating point arithmetic and reformulates the algorithms. This approach has been successful for a small number of problems only, for example, two–dimensional convex hulls [LM92], line arrangements [FM91] and Delaunay triangulations [For92]. The other approach uses exact real arithmetic. Although it has been known for a long time [Mig92, Yap94] that exact computation with algebraic numbers[1] is possible, it has also been felt that the overhead would be prohibitive. Serious experimental studies appeared only recently. In [KLN91, Fv93, BJMM93, MN94b] it is shown that the overhead for exact rational arithmetic is small in the context of geometric computations. In particular, floating point filters as introduced in [Fv93] are very effective. A floating point filter first approximates a computation using **doubles**. Only if a test cannot be performed conclusively on the basis of the **double** approximation, exact arithmetic is used as a back–up. In [BMS94] the same idea is generalized to a computation involving non–rational algebraic numbers, and in [Yap93, YD95] it is argued that it can be applied to all numerical computations arising in computational geometry. In this paper we define and implement the LEDA data type **real**. Any integer is a **real** and the class of **real**s is closed under addition, subtraction, multiplication, division and squareroot. The full repertoire of relational operators $(=, \neq, <, \leq, >, \geq,)$ is available on **real**s. The outcome is *exact*. In addition, arbitrary floating point approximations of **real** numbers can be computed. We define the class **real** in section 2 and also give two examples of the use of **real** numbers: an incircle test arising in the computation of Voronoi diagrams of line segments and the computation of line segment intersection. Starting in section 7 we give the implementation of **real**s. Our implementation is similar to the proposal by Yap and Dube [YD95]. In section 3 we review some facts about floating point computation.

[1]A real number is algebraic if it is the root of a polynomial with integer coefficients.

2. The manual pages of data type real

1. Definition

An instance x of the data type real is an algebraic real number. There are many ways to construct a real: either by conversion from *double*, *bigfloat*, *integer* or *rational* or by applying one of the arithmetic operators $+, -, *, /$ or $\sqrt{}$ to real numbers. One may test the sign of a real number or compare two real numbers by any of the comparison relations $=, \neq, <, \leq, >$ and $\geq$. The outcome of such a test is *exact*. There is also a non–standard version of the sign function: the call $x.sign(integer\ q)$ computes the sign of x under the precondition that $|x| \leq 2^{-q}$ implies $x = 0$. This version of the sign function allows the user to assist the data type in the computation of the sign of x, see the example below.

There are several functions to compute approximations of reals. The calls $tobigfloat(x)$ and $x.get\_bigfloat\_error()$ return *bigfloat*s $xnum$ and $xerr$ such that $|xnum - x| \leq xerr$. The user may set a bound on $xerr$. More precisely, after the call $x.improve\_approximation\_to(integer\ q)$ the data type guarantees $xerr \leq 2^{-q}$. One can also ask for *double* approximations of a real number x. The calls $todouble(x)$ and $x.get\_double\_error()$ return *double*s $xnum$ and $xerr$ such that $|xnum - x| \leq |xnum| * xerr$. Note that $xerr = \infty$ is possible.

2. Creation

reals may be constructed from data types *double*, *bigfloat*, *long*, *int* and *integer*. The default constructor $real()$ initializes the real to zero.

3. Operations

double	todouble(*real*)	returns the current double approximation of x.
bigfloat	tobigfloat(*real*)	returns the current bigfloat approximation of x.
double	x.get_double_error()	returns the relative error of the current double approximation of x, i.e., $\|x - todouble(x)\| \leq x.get\_double\_error() \cdot \|todouble(x)\|$.
bigfloat	x.get_bigfloat_error()	returns the absolute error of the current bigfloat approximation of x, i.e., $\|x - tobigfloat(x)\| \leq x.get\_bigfloat\_error()$.
int	sign(*real* x)	returns the sign of (the exact value of) x.
int	x.sign(*integer* q)	as above. *Precondition:* if $\|x\| \leq 2^{-q}$ then $x = 0$.

void x.improve_approximation_to(*integer* q)

(re-)computes the approximation of x such that $x.get\_bigfloat\_error() \leq 2^{-q}$ after the call $x.improve\_approximation\_to(q)$.

void x.compute_with_precision(*long* k)

(re-)computes the approximation of x; each numerical operation is carried out with k binary places.

void x.guarantee_relative_error(*long* k)

(re-)computes an approximation of x such that the relative error of the approximation is less than 2^{-k}.

ostream& ostream& $O \ll x$

writes the closest interval that is known to contain x to the output stream O

istream& istream& $I \gg$ *real&* x

reads *real* number x from the output stream I (in *double* format)

real abs(*real* x) absolute value of x

real sq(*real* x) square of x

real dist(*real* x, *real* y)

euclidean distance of vector (x,y) to the origin

real powi(*real* x, *int* n)

n.th power of x

4. Implementation

A real is represented by the expression which defines it and a *double* approximation $\hat{x}$ together with a relative error bound ϵ_x. The arithmetic operators $+, -, *, \sqrt{}$ take constant time. When the sign of a real number needs to be determined, the data type first computes a number q, if not already given as an argument to *sign*, such that $|x| \leq 2^{-q}$ implies $x = 0$. The bound q is computed as described in [Mig92]. Using *bigfloat* artihmetic, the data type then computes an interval I of maximal length 2^{-q} that contains x. If I contains zero, then x itself is equal to zero. Otherwise, the sign of any point in I is returned as the sign of x.

Two shortcuts are used to speed up the computation of the sign. Firstly, if the *double* approximation already suffices to determine the sign, then no *bigfloat* approximation is computed at all. Secondly, the *bigfloat* approximation is first computed only with small precision. The precision is then doubled until either the sign can be decided (i.e., if the current approximation interval does not contain zero) or the full precision 2^{-q} is reached.

5. Example

We give two examples of the use of the data type real. The examples deal with the Voronoi diagram of line segments and the intersection of line segments, respectively.

The following incircle test arises in the computation of Voronoi diagrams of line segments [BMS94]. For i, $1 \leq i \leq 3$, let $l_i : a_i x + b_i y + c_i = 0$ be a line in two–dimensional space and let $p = (0,0)$ be the origin. In general, there are two circles passing through p and touching l_1 and l_2. The centers of these circles have homogeneuos coordinates (x_v, y_v, z_v), where

$$
\begin{aligned}
x_v &= a_1c_2 + a_2c_1 \pm \operatorname{sign}(s)\sqrt{2c_1c_2(\sqrt{N} + D)} \\
y_v &= b_1c_2 + b_2c_1 \pm \operatorname{sign}(r)\sqrt{2c_1c_2(\sqrt{N} - D)} \\
z_v &= \sqrt{N} - a_1a_2 - b_1b_2
\end{aligned}
$$

and

$$
\begin{aligned}
s &= b_1|a_2| - b_2|a_1|, & N &= (a_1^2 + b_1^2)(a_2^2 + b_2^2) \\
r &= a_1|b_2| - a_2|b_1|, & D &= a_1a_2 - b_1b_2.
\end{aligned}
$$

Let us concentrate on one of these (say, we take the plus sign in both cases). The test whether l_3 intersects, touches or misses the circle amounts to determining the sign of

$$E := dist^2(v, l_3) - dist^2(v, p) = \frac{(a_3x_v + b_3y_v + c_3)^2}{a_3^2 + b_3^2} - (x_v^2 + y_v^2)$$

We show how to compute the sign of $\tilde{E} := (a_3^2 + b_3^2) \cdot E$ using our data type real. We assume that all coefficients a_i, b_i and c_i, $1 \leq i \leq 3$, are k bit integers, i.e., integers whose absolute value is bounded by $2^k - 1$. In [BMS94] we showed that for $\tilde{E} \neq 0$ we have $|\tilde{E}| \geq 2^{-24k-26)}$. These observations suggest the following program.

```
int INCIRCLE(int k, real a1, real b1, real c1, real a2, real b2, real c2, real a3, real
b3, real c3 )
{
  real RN = sqrt((a1 * a1 + b1 * b1) * (a2 * a2 + b2 * b2));
  real A = a1 * c2 + a2 * c1;
  real B = b1 * c2 + b2 * c1;
  real C = 2 * c1 * c2;
  real D = a1 * a2 - b1 * b2;
  real s = b1 * fabs(a2) - b2 * fabs(a1);
  real r = a1 * fabs(b2) - a2 * fabs(b1);
  int sign_x = sign(s);
  int sign_y = sign(r);
```

```
real x_v = A + sign_x * sqrt(C * (RN + D));
real y_v = B - sign_y * sqrt(C * (RN - D));
real z_v = RN - (a_1 * a_2 + b_1 * b_2);
real P = a_3 * x_v + b_3 * y_v + c_3 * z_v;
real D_3^2 = a_3 * a_3 + b_3 * b_3;
real R^2 = x_v * x_v + y_v * y_v;
real E = P * P - D_3^2 * R^2;
return E.sign(24 * k + 26);
}
```

We turn to the line segment intersection problem next. Assume that all endpoints have k–bit integer homogeneous coordinates. This implies that the intersection points have homogeneous coordinates (X, Y, W) where X, Y and W are (4 k + 3) - bit integers. The Bentley–Ottmann plane sweep algorithm for segment intersection [MN94a] needs to sort points by their x–coordinates, i.e., to compare fractions X_1/W_1 and X_2/W_2 where X_1, X_2, W_1, W_2 are as above. This is tantamount to determining the sign of the $8k + 7$ bit integer $X_1 * W_2 - X_2 * W_1$. If all variables X_i, W_i are declared **real** then their sign test will be performed quite efficiently. First, a *double* approximation is computed and then, if necessary, *bigfloat* approximations of increasing precision. In many cases, the *double* approximation already determines the sign. In this way, the user of the data type *real* gets the efficiency of a floating point filter [Fv93, MN94b] without any work on his side. This is in marked contrast to [Fv93, MN94b] and will be incorporated into [MN94a].

3. A Short Review of Floating Point Arithmetic

We review some facts about double precision floating point arithmetic according to the IEEE standard [IEE87]. A **double** is specified by a sign $s \in \{0,1\}$, an exponent $e \in [0 \ldots 2047]$ and a binary fraction $f = f_1 f_2 \ldots f_{52}$ with $f_i \in \{0,1\}$ for all $i, 1 \le i \le 52$. We also use f to denote the number $\sum_i f_i 2^{-i}$. The number v represented by a triple (s,e,f) is defined as follows:

- if $e = 2047$ and $f \neq 0$ then no number is represented (v is a NaN)
- if $e = 2047$ and $f = 0$ then $v = (-1)^s \cdot \infty$
- if $0 < e < 2047$ then $v = (-1)^s \cdot (1+f) \cdot 2^{e-1023}$
- if $e = 0$ and $f \neq 0$ then $v = (-1)^s \cdot f \cdot 2^{-1022}$ and v is called a denormalized number
- if $e = 0$ and $f = 0$ then $v = (-1)^s \cdot 0$ (= signed zero)

Let $MinDbl = 2^{-1022}$ and $MaxDbl = (2-2^{-52}) \cdot 2^{1023}$. We call a real number *approximable* if either $a = 0$ or $MinDbl \le |a| < MaxDbl + 2^{-53} \cdot 2^{1023}$. For any approximable number a let $round(a)$ be the floating point number nearest to a. If there are two floating point numbers equally near to a then $round(a)$ is the one where the least significant bit is zero. For any of the arithmetic operations $+, -, *, /, \sqrt{}$, the IEEE standard guarantees the following. Let z be the *exact* result of an operation applied to floating point operands.

- if z is approximable then $round(z)$ is delivered,
- if $|z| \ge MaxDbl + 2^{-53} \cdot 2^{1023}$ then an overflow exception is signaled and $\pm\infty$ is delivered where the sign is determined by z,
- if $0 < |z| < MinDbl$ then the result may be zero, a denormalized number or $\pm MinDbl$ with the restriction that z and the delivered result do not have opposite signs[2]; in some cases the underflow is signaled,
- if any argument of an operation is infinity or not a number, then the result is guaranteed to be infinity or not a number.

We need a relation between an approximable number a and its floating point approximation $round(a)$. Assume $a \neq 0$ and let e be such that $2^e \le |a| \le 2^{e+1}$. Then $round(a) = a + \rho(a)$ where $|\rho(a)| \le 2^{-53} \cdot 2^e$ and $2^e \le |round(a)| < 2^{e+1}$. Thus

$$\frac{|a - round(a)|}{|a|} \le \frac{2^{-53} \cdot 2^e}{2^e} = 2^{-53}$$

and

$$\frac{|a - round(a)|}{|round(a)|} \le \frac{2^{-53} \cdot 2^e}{2^e} = 2^{-53}.$$

The quantity 2^{-53} is so important that it deserves a special name. We call it the *precision* of floating point arithmetic and use *eps* to denote it. By the two bounds derived above we have

[2]The standard is actually more specific but this is of no importance for this paper

$$round(a) \in [(1-eps)\cdot a, (1+eps)\cdot a] \subset [a/(1+eps), a\cdot(1+eps)]$$

and

$$a \in [(1-eps)\cdot round(a), (1+eps)\cdot round(a)]$$

for any non–zero approximable number a.

We will frequently derive expressions for error bounds. These expressions involve the operators $+, *, /$ and have non–negative operands. When such an expression is evaluated using floating point arithmetic, further error incurs which must be compensated by post–multiplying with a *correction factor*. We now show how to compute this factor. We first determine for any expression E a number $ind(E)$ such that

$$\hat{E} \in [E/(1+eps)^{ind(E)}, E\cdot(1+eps)^{ind(E)}]$$

where $\hat{E}$ is the value of E computed in floating point arithmetic. We assume first that no underflow or overflow occurs in the computation of $\hat{E}$. This implies that all intermediate results are approximable. If E is an operand then $ind(E)=0$, if $E=E_1+E_2$ then ($\oplus$ denotes floating point addition)

$$\hat{E} = \hat{E}_1 \oplus \hat{E}_2 \geq \frac{\hat{E}_1+\hat{E}_2}{1+eps} \geq \frac{E_1/(1+eps)^{ind(E_1)} + E_2/(1+eps)^{ind(E_2)}}{1+eps}$$

and hence $ind(E_1+E_2) = 1 + max(ind(E_1), ind(E_2))$, and if $E = E_1 \cdot E_2$ or $E = E_1/E_2$ then

$$\hat{E} = \hat{E}_1 \odot \hat{E}_2 \geq \frac{\hat{E}_1\cdot\hat{E}_2}{1+eps} \geq \frac{E_1\cdot E_2}{(1+eps)^{1+ind(E_1)+ind(E_2)}}$$

and hence $ind(E_1\cdot E_2) = 1 + ind(E_1) + ind(E_2)$ and analoguously $ind(E_1/E_2) = 1 + ind(E_1) + ind(E_2)$. The upper bound for $\hat{E}$ can be shown by similar reasoning. We now choose our correction factor as $1 + k\cdot eps$ where k is a power of 2 and $(1+eps)^{ind(E)+1} \leq 1 + k\cdot eps$. Then

$$E \leq (1+eps)^{ind(E)}\cdot\hat{E} \leq \frac{(1+k\cdot eps)\cdot\hat{E}}{1+eps} \leq (1 \oplus k \odot eps)\odot\hat{E},$$

where the last inequality follows from the fact that $1 \oplus k \odot eps = 1 + k\cdot eps$ since k is a power of two. For small values of $ind(E)$ one can choose k to be the smallest power of two with $ind(E)+2 \leq k$.

What happens when there is overflow or underflow in the evaluation of E? We detect overflow by inspecting the final result; it must be infinity or not a number. So assume that the evaluation of some subexpression E_s underflows. Let us also assume that there is no division operator on the path from the subexpression E_s to the root of the expression E (in our error bounds this will be the case for any subexpression that can underflow). Then we may conceptually write E as $E = E_{a(s)}\cdot E_s + E_{b(s)}$ since any expression over $+$ and $*$ is a linear function of each of its subexpressions. Let $M_{a(s)}$ be a bound on $E_{a(s)}$. The value of $\widehat{E_s}$ is less than $MinDbl$ and due to underflow it may be set to zero. The underflow of E_s can therefore decrease the value of E by no more than $MinDbl\cdot M_{a(s)}$. Summing over all subexpressions we conclude that underflow can decrease the value of E

by no more than $MinDbl \cdot \sum_s M_{a(s)}$ where s ranges over all subexpressions of E. This implies that we have $E \le (1+eps)^{ind(E)} \cdot \hat{E} + MinDbl \cdot \sum_s M_{a(s)}$ where $\hat{E}$ is the computed value (now with the possibility of underflow). In our applications, we will always have $\hat{E} \ge eps$. In that case, if we guarantee that $MinDbl \cdot (\sum_s M_{a(s)}/eps) \le eps/2$ and choose k such that $(1+eps)^{ind(E)+1} + (1+eps) \cdot eps/2 \le 1 + k \cdot eps$ then

$$\begin{aligned}
E &\le (1+eps)^{ind(E)} \cdot \hat{E} + MinDbl \cdot \sum_s M_{a(s)} \\
&\le ((1+eps)^{ind(E)} + (MinDbl \cdot \sum_s M_{a(s)})/eps) \cdot \hat{E} \\
&\le ((1+eps)^{ind(E)} + eps/2) \cdot \hat{E} \\
&\le (1 + k \cdot eps) \cdot \hat{E}/(1+eps) \le (1 \oplus k \odot eps) \odot \hat{E}
\end{aligned}$$

Let us consider the example $E = eps + |x/z| \cdot \epsilon_x + |y/z| \cdot \epsilon_y$. The expression E has index 4 and the two multiplications and the two divisions may underflow. The sum of the multipliers of the "dangerous" subexpressions is $M := \epsilon_x + \epsilon_y + 1 + 1$. If ϵ_x and ϵ_y are both bounded by 2^{915} then clearly $MinDbl \cdot M \le eps^2/2$ and hence we may use the correction factor $1 + 8 \cdot eps$. In fact, *the factor* $1 + 8 \cdot eps$ *suffices for all bounds in this paper provided that incoming error bounds are bounded by* 2^{915}. We define both quantities as global constants.

⟨ Global functions and constants 3 ⟩ ≡
```
double pow2(long);
const double eps = pow2(−53);
const double correction = 1 + 8 ∗ eps;
const double MaxError = pow2(915);
const double MinDbl = pow2(−1022);
const double twoMinDbl = pow2(−1021);
const double MaxDbl = (2 − pow2(−52)) ∗ pow2(1023);
const double Infinity = pow2(1024);
```
See also chunks 5, 6, and 29.
This code is used in chunk 13.

4. Bigfloats.

LEDA's **bigfloat**s are a generalization of **double**s. The differences are:

- The exponent range is the set of all integers (type **integer**) and hence there is no underflow and overflow. This implies that all numbers are approximable and that $round(a) = 0$ implies $a = 0$.
- The mantissa length can be prescribed by the user. There are two ways to do this. One way is to call the function **bigfloat**::*set_glob_prec*(**int** p) to set the mantissa length to p. The other way is to call $add(x, y, p)$ for the addition of **bigfloat**s x and y with mantissa length p, and analogously *sub*, *mul*, *div* for subtraction, multiplication and division, respectively[3].

[3]The motivation for this syntax is to be able to evaluate different operations in a large expression with different precisions.

5. We need functions to decide whether **integer**s and **bigfloat**s are exactly representable by **double**s.

$\langle$ Global functions and constants 3 $\rangle$ +≡
int *isdouble*(**const integer** &x)
{ **return** ($x \equiv$ **integer**(*todouble*(**bigfloat**(x)))); }
int *isdouble*(**const bigfloat** &x)
{ **return** ($x \equiv$ **bigfloat**(*todouble*(x))); }

6. Also we use a constant for zero as an **integer**.

$\langle$ Global functions and constants 3 $\rangle$ +≡
const integer *zero_integer*(0);

7. The implementation

The implementation defines the files real.h and real.c. The first file contains the interface of class **real** and must be included into any program using **real**s. The latter file can be compiled into real.o. Any application of **real**s must specify real.o in the linking phase.

The public functions of class **real** are exactly the ones defined in the manual.

```
⟨real.h 7⟩ ≡
#ifndef REAL_H
#define REAL_H
#include "/KM/usr/burnikel/cweb/bigfloat/new/bigfloat.h"
#include <LEDA/rational.h>
  class real_rep;
  class real
  {
    ⟨private data and functions of type real 15⟩
  public:
    real( );
    real(double);
    real(int);
    real(const integer &);
    real(const bigfloat &);
    real(const rational &);
    real(const real &);
    real(real_rep &);
    ~real( );
    real &operator=(const real &);
    friend double todouble(const real &);
    inline double todouble( ) const { return ::todouble(*this); }
    friend bigfloat tobigfloat(const real &);
    inline bigfloat tobigfloat( ) const { return ::tobigfloat(*this); }
    double get_double_error( ) const;
    bigfloat get_bigfloat_error( ) const;
    inline friend int sign(const real &x) { return x.sign( ); }
    int sign( ) const;
    int sign(const integer &q) const;
    int sign(long) const;
    void improve_approximation_to(const integer &q) const;
    void compute_with_precision(long k) const;
    void guarantee_relative_error(long k) const;
    friend ostream &operator<<(ostream &O, const real &x);
    friend istream &operator>>(istream &I, real &x);
    ⟨Comparison operators 8⟩
    ⟨Arithmetic operators 9⟩
  };
```

$\langle$ LEDA functions 11 $\rangle$
$\langle$ Assignment operators 10 $\rangle$
$\langle$ Declaration of mathematical functions in real.h 12 $\rangle$
#endif

8. We have the full list of comparison operators.

$\langle$ Comparison operators 8 $\rangle \equiv$

```
friend int operator < (const real &, const real &);
friend int operator <= (const real &, const real &);
friend int operator > (const real &, const real &);
friend int operator >= (const real &, const real &);
friend int operator == (const real &, const real &);
friend int operator != (const real &, const real &);
```

This code is used in chunks 7 and 17.

9. All arithmetic operators are friends of classes **real** and **real_rep**.

$\langle$ Arithmetic operators 9 $\rangle \equiv$

```
friend real operator+(const real &, const real &);
friend real operator-(const real &, const real &);
friend real operator*(const real &, const real &);
friend real operator/(const real &, const real &);
friend real operator-(const real &);
friend real sqrt(const real &);
```

This code is used in chunks 7 and 17.

10. We also have some fancy assignment operators.

$\langle$ Assignment operators 10 $\rangle \equiv$

```
inline real operator+=(real &x, const real &y)
{ x = x + y; return x; }
inline real operator-=(real &x, const real &y)
{ x = x - y; return x; }
inline real operator*=(real &x, const real &y)
{ x = x * y; return x; }
```

This code is used in chunk 7.

11. The following functions have to be defined for every LEDA type.

$\langle$ LEDA functions 11 $\rangle \equiv$

```
inline void Print(const real &x, ostream &out) { out << x; }
inline void Read(real &x, istream &in) { in >> x; }
inline int compare(const real &x, const real &y) { return (x - y).sign(); }
inline char *Type_Name(const real *) { return "real"; }
```

This code is used in chunk 7.

12. Mathematical functions. This list is still small.

⟨ Declaration of mathematical functions in real.h 12 ⟩ ≡
real *abs*(**const real** &*x*);
real *sq*(**const real** &*x*);
real *dist*(**const real** &*x*, **const real** &*y*);
real *powi*(**const real** &*x*, **int** *n*);

This code is used in chunk 7.

13. The file real.c has a similar structure as the file real.h. For each group of functions there is a section containing the implementation. In addition, we define some constants and the class **real_rep**.

⟨ `real.c` 13 ⟩ ≡
#include `"real.h"`
#include `<math.h>`
⟨ Global functions and constants 3 ⟩
⟨ Basic declarations 16 ⟩
⟨ Declaration of class **real_rep** 17 ⟩
⟨ Constructors and destructors 18 ⟩
⟨ Basic functions 23 ⟩
⟨ Operators 32 ⟩
⟨ Sign functions 50 ⟩
⟨ Functions for improving the approximation 65 ⟩
⟨ Implementation of mathematical functions in real.c 30 ⟩

14. Realisation of class real.

A **real** may be exactly representable by a **double**. In this case, we call it a **basic real**. Otherwise, if a **real** is not (known to be) a **double**, it is defined by an expression tree whose leaves are given by basic **real**s. In that case we maintain the expression tree and the real only needs to know a pointer to the root of the tree. The nodes of expression trees are realized by the class **real_rep** which is the work horse of our implementation. The member functions and all operations, except the arithmetic operators, are first implemented on **real_rep**s and then lifted to **real**s. The design to separate an object of type **real** from its tree representation has the following advantages:

- Copying a **real** takes constant time since it basically amounts to copying a pointer to a **real_rep**.
- Since reals may share subtrees, we have to provide our own memory management. The part of the information that can be automatically allocated is stored in class **real** itself, and the dynamically allocated part is stored in class **real_rep**.

15. The private data of class **real** is organized as follows.

- If *is_double* is true then the **real** is basic and the value is stored in the member *value*. If the **real** is part of an expression tree then *PTR* is defined and points to a **real_rep** also containing the value. Storing the value twice simplifies programming; of course, it also takes extra space.

- If *is_double* is *false*, *PTR* points to the root of an expression tree defining the **real**; the content of *value* is meaningless in this case.

⟨ private data and functions of type real 15 ⟩ ≡

```
bool is_double;
double value;
real_rep *PTR;
```

See also chunk 25.

This code is used in chunk 7.

16. The class **real_rep**.

A **real_rep** object is either a leaf, i.e., represents a **double** number or a **bigfloat** number, or it is constructed by an arithmetic operation from two other **real_rep** objects. We define the enumeration type *constructortype* to distinguish these cases.

A **real_rep** may contain a **bigfloat** approximation to the exact value of the represented number. Since this approximation is space–consuming and not always needed we encapsulate it in a structure called *bf_app*. The structure is organized as follows.

- *NUM_BF* is the **bigfloat** approximation,
- *ERROR_BF* bounds the error of the approximation *NUM_BF*, i.e., $|NUM\_BF - NUM\_EXACT| \leq ERROR\_BF$.

⟨ Basic declarations 16 ⟩ ≡

```
enum constructortype {
  DOUBLE = 0, BIGFLOAT = 1, NEGATION = 2, SQUAREROOT = 3,
      ADDITION = 4, SUBTRACTION = 5, MULTIPLICATION = 6,
      DIVISION = 7
};
struct app_bf {
  bigfloat NUM_BF;
  bigfloat ERROR_BF;
  app_bf( ) { }
  app_bf(const bigfloat &x, const bigfloat &error_x = pZero)
  { NUM_BF = x; ERROR_BF = error_x; }
};
```

See also chunks 52 and 83.

This code is used in chunk 13.

17. We can now define the class **real_rep**. Any object of class **real_rep** contains a **double** approximation *NUM*, a **double** *ERROR*, a pointer *APP_BF_PTR* to a **bigfloat** approximation (of type **app_bf**), the **constructortype** *CON*, pointers *OP1* and *OP2* to the two operands, and a *count* of the number of pointers to the object. Furthermore there is a long member *status* that we use for efficient searching through the expression trees and a pointer *sep_bound_ptr* to a structure of type *sep_bound* that we define in the section about sign computation.

We will always use NUM_EXACT to denote the exact value of a **real_rep**. The quantities NUM, $ERROR$, NUM_BF and $ERROR\_BF$ satisfy the following inequalities: Either $ERROR = \infty$ or $ERROR \le MaxError$, $NUM = 0$ or $MinDbl \le NUM \le MaxDbl$ (i.e., NUM is not a denormalized number) and

$$|NUM\_EXACT - NUM| \;\le\; |NUM| \cdot ERROR.$$

If $APP\_BF\_PTR \ne nil$ then

$$|NUM\_EXACT - NUM\_BF| \;\le\; ERROR\_BF < \infty.$$

Note that $ERROR$ is the *relative* error of the **double** approximation and $ERROR\_BF$ is the *absolute* error of the **bigfloat** approximation. Also note that the **bigfloat** approximation may not exist (if $APP\_BF\_PTR \equiv nil$) and that some numbers have no **double** approximation (if $ERROR \equiv \infty$). However, if the **bigfloat** approximation of a **real_rep** is defined, then it is also defined for its children.

If a **real** has a **double** and a **bigfloat** approximation then the **double** approximation is *essentially* the conversion of the **bigfloat** value to **double**. Since our algorithms do not exploit any connection between the **double** and the **real** approximation we formalize no connection.

The member functions of class **real_rep** do the following:

- *init_app_bf*( ) initializes the **bigfloat** approximation,
- *adjust_dbl*( ) adjusts the **double** approximation to an improved **bigfloat** approximation,
- *compute_op*(**int** p) computes the last operation (given by member CON) in precision p, using bigfloat arithmetic,
- *compute_approximation* implements the function **real** :: *compute_with_precision*,
- *sign_with_separation_bound*(**int** k, **bool** *supported*) returns the sign of the represented number x; if the boolean parameter *supported* is true, the sign computation is supported by the information that $|x| \le 2^k$ implies $x = 0$,
- *guarantee_bound_two_to*(**integer** e) improves the **bigfloat** approximation of the number such that $ERROR\_BF$ is bounded by 2^e,
- *num_bf*( ) and *error_bf*( ) return references to the values $APP\_BF\_PTR{\to}NUM\_BF$ and $APP\_BF\_PTR{\to}ERROR\_BF$,
- *num_exp*( ) and *err_exp*( ) compute the binary logarithms of nonzero NUM_BF and $ERROR\_BF$, rounded up towards the next integer,
- *exact*( ) tells whether $ERROR\_BF$ is zero.

⟨ Declaration of class **real_rep** 17 ⟩ ≡

struct sep_bound;
class real_rep {
friend class real;

```
double NUM;
double ERROR;
app_bf *APP_BF_PTR;
constructortype CON;
real_rep *OP1, *OP2;
int count;
long status;
sep_bound *sep_bound_ptr;
real_rep( );
real_rep(double);
real_rep(const bigfloat &);
~real_rep( );
LEDA_MEMORY(real_rep)
void init_app_bf( );
friend double todouble(const real &);
friend bigfloat tobigfloat(const real &);
void adjust_dbl( );
void compute_op(long);
void compute_approximation(long);
int sign_with_separation_bound (const integer &, bool = false ) ;
void guarantee_bound_two_to(const integer &);
inline bigfloat &num_bf( ) { return APP_BF_PTR->NUM_BF; }
inline bigfloat &error_bf( ) { return APP_BF_PTR->ERROR_BF; }
inline bool exact( ) { return isZero(error_bf( )); }
inline integer num_exp( ) { return log2(num_bf( )); }
inline integer err_exp( ) { return log2(error_bf( )); }
⟨Local functions of class real_rep 53⟩
friend ostream &operator<<(ostream &, const real &);
⟨Arithmetic operators 9⟩
⟨Comparison operators 8⟩
} ;
```

This code is used in chunk 13.

18. Constructors and destructors

We start with the constructors for **class real**. The first three construct basic **real**s from an empty argument list, from a **double**, and from an **int**. They are easy to implement. The next two – for **integer** and **bigfloat** – construct **real**s that in general are not representable by **double**s but are defined by a single **real_rep** node. Rationals are constructed differently. We first construct **real**s for numerator and denominator, respectively, and then return their **real** division. Hence the result is a pointer to a tree with three nodes unless the rational is a **double**. The constructor that takes a **real_rep** argument simply assigns the adress of the **real_rep** to PTR and sets *is_double* ≡ *false*.

⟨ Constructors and destructors 18 ⟩ ≡

```
real::real() { is_double = true; value = 0; PTR = nil; }
real::real(int x) { is_double = true; value = x; PTR = nil; }
real::real(double x)
{ is_double = true; value = x; PTR = nil; }
real::real(const bigfloat &x)
{
  if (is_double = isdouble(x)) {
    value = ::todouble(x);
    PTR = nil;
  }
  else PTR = new real_rep (x);
}
real::real(const integer &x)
{
  if (is_double = isdouble(x)) {
    value = x.todouble();
    PTR = nil;
  }
  else PTR = new real_rep (bigfloat(x));
}
real::real(const rational &x)
{
  real OP1 = *new real_rep (x.numerator());
  real OP2 = *new real_rep (x.denominator());
  *this = OP1/OP2;
}
real::real(real_rep &x_rep) { is_double = false; PTR = &x_rep; }
```

See also chunks 19, 20, and 21.

This code is used in chunk 13.

19. Copy constructor and assignment operator are treated in direct succession because they are closely related. Both functions distinguish the two possible cases of *is_double*. If the argument is not basic, the copy constructor simply copies the pointer to the **real_rep** of its arguments and records the fact that the number of **real**s sharing the representation has increased by one. The assignment operator does basically the same, but is difficult to implement correctly. Notice that we must take care that

1. an existing pointer $x.PTR \neq nil$ is *not* copied for $x.is\_double \equiv true$
2. an existing reference PTR is correctly deleted
3. everything remains correct even for $x.PTR \equiv PTR$ or even $*$**this** $= x$

In addition, if PTR pointed to some **real_rep** node, it decreases the count of this node by one and deletes the node if the count goes to zero.

⟨ Constructors and destructors 18 ⟩ +≡

```
real::real(const real &x)
{
  if (is_double = x.is_double) { value = x.value; PTR = nil; }
  else { x.PTR→count++; PTR = x.PTR;
  }
}
real &real::operator=(const real &x)
{     /* we avoid problems by ruling out the assignment x = x; */
  if (this ≡ &x) return (*this);     /* first the actions that must anyway be
          performed, in the respective cases of is_double */
  if (is_double = x.is_double) value = x.value;
  else x.PTR→count++;     /* second, an existing reference in PTR must always be
          deleted; if x.PTR ≡ PTR ≠ nil and is_double ≡ false, this leaves the number
          of references to *PTR unchanged, because we increased x.PTR before, and
          for is_double ≡ true we in fact lose one reference because PTR is set to nil
          at the end */
  if (PTR) if (−−PTR→count ≡ 0) delete PTR;     /* last, we set this→PTR */
  if (is_double) PTR = nil; else PTR = x.PTR;
  return (*this);
}
```

20. Beside the constructors for class **real** we need three constructors for class **real_rep**. The first one is the straightforward empty constructor. The second one takes **double**s and is only needed when one of the **real** operators needs to initialize a leaf representing a basic **real**. The third constructor takes **bigfloat**s and is used by the corresponding **real** :: constructor. Note that even in this case we have to initialize the **double** approximation by calling the function *adjust_dbl*( ). It is defined in section 24.

⟨ Constructors and destructors 18 ⟩ +≡

```
real_rep::real_rep()
{ APP_BF_PTR = nil; sep_bound_ptr = nil; count = 1; status = 0; }
real_rep::real_rep(double x)
{ APP_BF_PTR = nil; sep_bound_ptr = nil;
  OP1 = OP2 = nil; CON = DOUBLE;
  NUM = x; ERROR = 0;
  count = 1; status = 0;
}
```

real_rep :: **real_rep**(**const bigfloat** &x)
{
  APP_BF_PTR = **new app_bf** (x, 0); *sep_bound_ptr* = *nil*;
  OP1 = *OP2* = *nil*; *CON* = *BIGFLOAT*;
  adjust_dbl( );
  count = 1; *status* = 0;
}

21. We discuss the destructors of classes **real** and **real_rep**. When a **real** is destroyed we simply decrease the counter of the corresponding **real_rep** (if it exists). When the counter reaches zero we also delete the **real_rep**. In order to delete a **real_rep** we free the storage allocated for the **real_rep** (this is done automatically except for the storage allocated for the optional **bigfloat** approximation and the memory addressed by *sep_bound_ptr*) and decrease the counters of the operands (if any) defining the **real_rep**. If any counter reaches zero we destroy the operands recursively.

⟨ Constructors and destructors 18 ⟩ +≡
  real :: ∼**real**( )
  { **if** (*PTR*) **if** (−− *PTR*→*count* ≡ 0) **delete** *PTR*;
  }
  real_rep :: ∼**real_rep**( )
  {
    if (*APP_BF_PTR*) **delete** *APP_BF_PTR*;
    if (*sep_bound_ptr*) **delete** *sep_bound_ptr*;
    if (*OP1* ∧ (−− *OP1*→*count* ≡ 0)) **delete** *OP1*;
    if (*OP2* ∧ (−− *OP2*→*count* ≡ 0)) **delete** *OP2*;
#**ifdef** *DEBUG_REAL*
    APP_BF_PTR = *nil*;
    OP1 = *OP2* = *nil*;
#**endif**
  }

22. Basic functions

Class **real_rep** has the basic functions *init_app_bf*, *adjust_dbl*, *num_exp* and *err_exp*, and the basic functions of class **real** are *todouble*, *get_dbl_error*, *tobigfloat*, *get_precision*, and the stream operators $\ll$ and $\gg$. We start with the basic functions of class **real_rep**.

23. The function *init_app_bf*( ) initializes the **bigfloat** approximation of a **real_rep** (if not defined already). If the **double** approximation is meaningful then we simply convert it to **bigfloat** format. Otherwise the functions calls itself recursively for the operands and computes the operation given by *CON* in precision 32.

$\langle$ Basic functions 23 $\rangle \equiv$

```
void real_rep::init_app_bf()
{
  if (APP_BF_PTR) return;
  if (OP1) OP1→init_app_bf();
  if (OP2) OP2→init_app_bf();
  if (ERROR < Infinity) {
    APP_BF_PTR = new app_bf (bigfloat(NUM),
        abs(bigfloat(ERROR * NUM * correction)));
    return;
  }
  APP_BF_PTR = new app_bf ();
  compute_op(53);
}
```

See also chunks 24, 26, 27, and 28.
This code is used in chunk 13.

24. The function *adjust_dbl*( ) adjusts the **double** approximation $(NUM, ERROR)$ to the current **bigfloat** approximation (which is guaranteed to exist when *adjust_dbl* is called). The adjustment is non–trivial since the **bigfloat** error bound is absolute and the **double** error bound is relative and since we have to take care of underflow. We need to distinguish several cases, two of which are simple. If $|NUM\_BF| > MaxDbl$ or $ERROR\_BF > MaxDbl$ then we simply set $ERROR$ to infinity and if $NUM\_BF = ERROR\_BF = 0$ then we set $NUM = ERROR = 0$. So assume that neither is the case. Let $n = NUM\_EXACT$, $nbf = NUM\_BF$, $ebf = ERROR\_BF$, $\hat{e} = \mathbf{double}(ERROR\_BF)$, and $\hat{n} = \mathbf{double}(NUM\_BF)$. Assume first that $\hat{n} \neq 0$. Then

$$\begin{aligned}
|\hat{n} - n| &\leq |\hat{n} - nbf| + |nbf - n| \\
&\leq eps \cdot |\hat{n}| + ebf \\
&\leq (eps + ebf/|\hat{n}|) \cdot |\hat{n}| \\
&\leq (eps + Max(MinDbl, (1 + eps) \cdot \hat{e})/|\hat{n}|) \cdot |\hat{n}|,
\end{aligned}$$

since $ebf \leq Max(MinDbl, (1 + eps)\hat{e})$. The formula above has index 4 (cf. section 3) and hence $1 + 8eps$ is a suitable correction factor for its floating point evaluation (this correction factor even allows to drop the multiplication of $\hat{e}$ by $(1 + eps)$).

Assume next that $\hat{n} = 0$. Then $|n| \le ebf + MinDbl \le (1 + eps) \cdot \hat{e} + 2MinDbl$. We set $NUM = \hat{e} \oplus 2MinDbl$. Then

$$|n - NUM| \le 2(1 + eps)^2 NUM \le 2 \odot (1 + 8eps) \odot NUM\,.$$

In either case, if *ERROR* exceeds *MaxError* (this will also be the case when the division $|\hat{e}|/|\hat{n}|$ overflows) we invalidate our computation and set *ERROR* to infinity.

⟨ Basic functions 23 ⟩ +≡

```
void real_rep::adjust_dbl( )
{
  double n_head = todouble(num_bf( ));
  double e_head = todouble(error_bf( ));
  if (n_head ≠ 0) {
    NUM = n_head;
    ERROR = eps + Max(MinDbl, e_head)/fabs(n_head);
  }
  else {
    NUM = e_head + 2 * MinDbl;
    ERROR = 2;
  }
  ERROR = ERROR * correction;
  if ((ERROR ≥ MaxError) ∨ (fabs(n_head) ≡ Infinity)) {
    NUM = 1; ERROR = Infinity;
  }
}
```

25. We declare a private function **real** :: *adjust* to adjust the state of a real. That is, we check whether a real is exactly a **double** and if so, we turn it into a basic **real**. In any case we adjust the **double** approximation. Not that we have to cast away the constancy of the **real** before changing its private data.

⟨ private data and functions of type real 15 ⟩ +≡

```
void adjust( ) const;
```

26. ⟨ Basic functions 23 ⟩ +≡

```
void real::adjust( ) const
{
  if (is_double) return;
  if (PTR→exact( ) ∧ isdouble(PTR→num_bf( ))) {
    ((real &) *this).is_double = true;
    ((real &) *this).value = ::todouble(PTR→num_bf( ));
    return;
  }
  PTR→adjust_dbl( );
}
```

27. We give out a **real** as an interval containing its exact value. The size of the interval depends on the quality of the **bigfloat** approximation.

We assume that **real** input reads from a **double** variable.

⟨ Basic functions 23 ⟩ +≡

```
ostream &operator<<(ostream &out, const real &x)
{
  bigfloat low_bound, upp_bound;
  long prec;
  if (x.is_double) low_bound = upp_bound = x.value;
  else {
    x.PTR->init_app_bf();
    prec = (log2(x.PTR->num_bf()) - log2(x.PTR->error_bf())).tolong();
    if (prec < 53) prec = 53;
    low_bound = sub(x.tobigfloat(), x.PTR->error_bf(), prec, TO_N_INF);
    upp_bound = add(x.tobigfloat(), x.PTR->error_bf(), prec, TO_P_INF);
  }
  out << "[" << low_bound << ",";
  out << upp_bound << "]";
  return out;
}
istream &operator>>(istream &in, real &x)
{ double x_num; in >> x_num;
  x = real(x_num); return in;
}
```

28. The other basic functions of class **real** do the obvious things.

⟨ Basic functions 23 ⟩ +≡

```
double todouble(const real &x)
{
  if (x.is_double) return x.value;
  if (x.PTR->APP_BF_PTR) x.PTR->adjust_dbl();
  return x.PTR->NUM;
}
double real::get_double_error() const
{
  if (is_double) return 0;
  if (PTR->APP_BF_PTR) PTR->adjust_dbl();
  return PTR->ERROR;
}
bigfloat tobigfloat(const real &x)
{
  if (x.is_double) return x.value;
  else {
    x.PTR->init_app_bf();
    return x.PTR->num_bf();
```

```
    }
  }
  bigfloat real::get_bigfloat_error() const
  {
    if (is_double) return 0;
    else {
      PTR→init_app_bf();
      return PTR→error_bf();
    }
  }
```

29. Now we list several auxiliary functions that we use in later sections. We need to compute

- $ldexp\_bf(x, k) = x \cdot 2^k$ for **bigfloat**s x and integers k,
- the maximum of two integers,
- the sign of a **double**.

⟨ Global functions and constants 3 ⟩ +≡

```
  inline bigfloat ldexp_bf(const bigfloat &x, const integer &k)
  { return bigfloat(x.get_significant(), x.get_exponent() + k); }
  integer Maximum(integer x, integer y) { return (x > y ? x : y); }
  int sign(double x)
  { if (x ≡ 0) return 0; else return (x > 0 ? 1 : −1); }
```

30. Mathematical functions.

We implement some of the most frequently used functions from "math.h" for type **real**.

⟨ Implementation of mathematical functions in real.c 30 ⟩ ≡

```
  real abs(const real &x) { return (x.sign() < 0 ? −x : x); }
  real sq(const real &x) { return x * x; }
  real dist(const real &x, const real &y) { return sqrt(x * x + y * y); }
  real powi(const real &x, int n)
  {
    real y = x, z = 1;
    int n_prefix = n;
    while (n_prefix > 0) {
      if (n_prefix % 2) z = z * y;
      n_prefix = n_prefix/2;
      y = y * y;
    }
    return z;
  }
```

This code is used in chunk 13.

31. The arithmetic operations

We give the implementation of the arithmetic operators $+, -, *, /$ and $\sqrt{}$. The implementations have a common structure. If both operands are basic, we check if the **double** operation already gives an *exact* result which may then be returned immediately. We believe that the time for checking this is well invested because, if the check succeeds

1. it saves space,
2. it saves the time for computing error bounds and doing dynamic memory allocation,
3. it may detect "degenerate" values without expensive **bigfloat** arithmetic.

If one of the operands is not a **double**, or if the result is not exactly representable in **double** format, we create **real_rep** nodes for the operands, if necessary, and also we create new node for the result z. There we record the type of the operation in the CON field, store pointers to the operands, increase the counters of the operands, and then compute the **double** approximation and the error bound for z.

Clearly, if either operand is not approximable (or no approximation is known), we cannot compute an approximation of z. So in our numerical analysis we will tacitly assume that both operands are approximable.

32. We start with the binary operations addition, subtraction, and multiplication.

⟨ Operators 32 ⟩ ≡

```
real operator+(const real &x, const real &y)
{
  ⟨ if addition is exact return double result 36 ⟩
  ⟨ create nodes for two operands 33 ⟩
  real_rep &z_rep = *new real_rep ( );
  ⟨ initialize z_rep as result of a binary operation 34 ⟩
  z_rep.CON = ADDITION;
  z_rep.NUM = x.PTR→NUM + y.PTR→NUM;
  ⟨ compute error for addition or subtraction 38 ⟩
  ⟨ if approximation is not valid set error to Infinity 35 ⟩
  return z_rep;
}
real operator−(const real &x, const real &y)
{
  ⟨ if subtraction is exact return double result 37 ⟩
  ⟨ create nodes for two operands 33 ⟩
  real_rep &z_rep = *new real_rep ( );
  ⟨ initialize z_rep as result of a binary operation 34 ⟩
  z_rep.CON = SUBTRACTION;
  z_rep.NUM = x.PTR→NUM − y.PTR→NUM;
  ⟨ compute error for addition or subtraction 38 ⟩
  ⟨ if approximation is not valid set error to Infinity 35 ⟩
  return z_rep;
}
```

real operator $*$ (**const real** $\&x$, **const real** $\&y$)
{
⟨ if multiplication is exact return double result 39 ⟩
⟨ create nodes for two operands 33 ⟩
real_rep $\&z\_rep = *$**new real_rep** ( );
⟨ initialize *z_rep* as result of a binary operation 34 ⟩
$z\_rep.CON = MULTIPLICATION$;
$z\_rep.NUM = x.PTR\text{→}NUM * y.PTR\text{→}NUM$;
⟨ compute error for multiplication 40 ⟩
⟨ if approximation is not valid set error to Infinity 35 ⟩
return *z_rep*;
}

See also chunks 41, 45, and 49.
This code is used in chunk 13.

33. We give the common parts of the implementations first. First we create nodes for the operands x and y, if not already existing. We simply have to use the **real_rep** constructor. Note that we have to cast away the constancy of x and y away before manipulating their data.

⟨ create nodes for two operands 33 ⟩ ≡
if $(\neg x.PTR)$ $(($**real** $\&)\ x).PTR =$ **new real_rep** $(x.value)$;
if $(\neg y.PTR)$ $(($**real** $\&)\ y).PTR =$ **new real_rep** $(y.value)$;
double $\&x\_num = x.PTR\text{→}NUM$;
double $\&y\_num = y.PTR\text{→}NUM$;

This code is used in chunks 32 and 41.

34. We initialize the result *z_rep*.

⟨ initialize *z_rep* as result of a binary operation 34 ⟩ ≡
$z\_rep.OP1 = x.PTR$; $z\_rep.OP2 = y.PTR$;
$z\_rep.OP1\text{→}count{+}{+}$; $z\_rep.OP2\text{→}count{+}{+}$;
double $\&NUM = z\_rep.NUM$;

This code is used in chunks 32 and 41.

35. Checking the validity of the **double** approximation is done as follows. If one of the incoming error bounds is *Infinity*, our new error bound cannot be valid. The same holds if the computed approximation $z\_rep.NUM$ is NaN or $\pm\infty$, or if the computed error bound $z\_rep.NUM$ is larger than $MaxError$ or NaN. Note that we do not need to check for the NaN cases separately because a comparison with a NaN always gives the answer "no".

⟨ if approximation is not valid set error to Infinity 35 ⟩ ≡
if $(\neg((fabs(NUM) + x.PTR\text{→}ERROR + y.PTR\text{→}ERROR < Infinity) \wedge (z\_rep.ERROR < MaxError)))$ {
$z\_rep.ERROR = Infinity$;
$NUM = 1$;
}

This code is used in chunks 32 and 41.

36. Now we skip to the addition–specific parts. It is possible to test efficiently whether a **double** addition $s = a + b$ is exact. In fact, one can show that – irrespective of which rounding mode is active or whether the operands are normalized or denormalized –, one of the two subtractions $(a \oplus b) \ominus b$ or $(a \oplus b) \ominus a$ is exact. This implies immediately that the addition is exact if and only if both of the following equalities hold[4] (which is easy to test):

$$(a \oplus b) \ominus b = a \wedge (a \oplus b) \ominus a = b.$$

$\langle$ if addition is exact return double result 36 $\rangle \equiv$

double *NUM_result*;
if $(x.is\_double \wedge y.is\_double)$ {
  $NUM\_result = x.value + y.value$;
  if $((NUM\_result - x.value \equiv y.value) \wedge ((NUM\_result - y.value) \equiv x.value))$
    return *NUM_result*;
}

This code is used in chunk 32.

37. The same test procedure applies to subtraction.

$\langle$ if subtraction is exact return double result 37 $\rangle \equiv$

double *NUM_result*;
if $(x.is\_double \wedge y.is\_double)$ {
  $NUM\_result = x.value - y.value$;
  if $((NUM\_result + y.value \equiv x.value) \wedge (x.value - NUM\_result \equiv y.value))$
    return *NUM_result*;
}

This code is used in chunk 32.

38. Now we give the error analysis for addition and subtraction. Because they are essentially the same, we only deal with addition.

Here and in the analysis of the operations to follow we use x and y to denote the operands, $\hat{x}$ and $\hat{y}$ to denote the **double** approximations of x and y and ϵ_x and ϵ_y to denote the relative error of $\hat{x}$ and $\hat{y}$. Thus $|x - \hat{x}| \leq \epsilon_x \cdot |\hat{x}|$ and $|y - \hat{y}| \leq \epsilon_y \cdot |\hat{y}|$. Let $\hat{z}$ denote the floating point approximation of the result of the floating point addition of $\hat{x}$ and $\hat{y}$. Assume first that $|\hat{z}| \geq MinDbl$. Then

$$\begin{aligned} |\hat{z} - (x + y)| &\leq |\hat{z} - (\hat{x} + \hat{y})| + |\hat{x} - x| + |\hat{y} - y| \\ &\leq |\hat{z}| \cdot eps + |\hat{x}| \cdot \epsilon_x + |\hat{y}| \cdot \epsilon_y \\ &\leq |\hat{z}| \cdot (eps + |\frac{\hat{x}}{\hat{z}}| \cdot \epsilon_x + |\frac{\hat{y}}{\hat{z}}| \cdot \epsilon_y) \end{aligned}$$

Dividing both sides of the inequality by $|\hat{z}|$ we obtain

$$\epsilon_z \leq eps + |\frac{\hat{x}}{\hat{z}}| \cdot \epsilon_x + |\frac{\hat{y}}{\hat{z}}| \cdot \epsilon_y \tag{1}$$

[4] The sufficiency of this condition follows immediately from [Knu80] Theorem 4.4.4 B

This formula has index 4. According to section 3 we may therefore use the correction factor $1 + 8 \cdot eps$ provided that ϵ_x and ϵ_y are both bounded by 2^{900}.

Assume next that $|\hat{z}| < MinDbl$. Then $|\hat{x} + \hat{y}| < MinDbl$ and hence

$$\begin{aligned} |z| &\le |x - \hat{x}| + |y - \hat{y}| + |\hat{x} + \hat{y}| \\ &\le |\hat{x}| \cdot \epsilon_x + |\hat{y}| \cdot \epsilon_y + MinDbl \\ &\le (1 + eps)^2 \cdot (|\hat{x}| \odot \epsilon_x \oplus |\hat{y}| \odot \epsilon_y) + 3MinDbl \\ &\le (1 + eps)^3 \cdot (|\hat{x}| \odot \epsilon_x \oplus |\hat{y}| \odot \epsilon_y \oplus 4MinDbl) \end{aligned}$$

where the term $2MinDbl$ accounts for the possible underflow in the multiplications. With $NUM = (|\hat{x}| \odot \epsilon_x \oplus |\hat{y}| \odot \epsilon_y \oplus 4MinDbl$ we have $|x + y - NUM| \le 2 \cdot (1 + eps)^3 |NUM|$. We may therefore set $ERROR = 2 * correction$.

```
⟨ compute error for addition or subtraction 38 ⟩ ≡
  if (fabs(NUM) ≥ MinDbl) {
    z_rep.ERROR = fabs(x_num/NUM) * x.PTR→ERROR + fabs(y_num/NUM) *
        y.PTR→ERROR + eps;
  }
  else {
    NUM = fabs(x_num)*x.PTR→ERROR+fabs(y_num)*y.PTR→ERROR+4*MinDbl;
    z_rep.ERROR = 2;
  }
  z_rep.ERROR *= correction;
```

This code is used in chunk 32.

39. We turn to multiplication–specific parts. Checking exactness of **double** multiplication seems more difficult than for addition. However, if both operands are "single–precision", that is, if the last 26 bits are zero, we know that the exact value of the multiplication is representable in **double** format. Hence in this case the **double** multiplication is guaranteed to compute exactly.

Note that the position of the **double**'s low word is machine–dependent. For example, for SUN SPARC's the *BIG_ENDIAN* byte ordering is active, an INTEL PC's use *LITTLE_ENDIAN* byte ordering.

```
⟨ if multiplication is exact return double result 39 ⟩ ≡
  double NUM_result;
  long *x_ptr, *y_ptr;
  if (x.is_double ∧ y.is_double) {
    NUM_result = x.value * y.value;
    x_ptr = (long *) &(x.value);
    y_ptr = (long *) &(y.value);
#ifndef LITTLE_ENDIAN
    x_ptr++; y_ptr++;
#endif
    if (¬(*x_ptr & #07ffffff) ∧ ¬(*y_ptr & #07ffffff)) {
      NUM_result = x.value * y.value;
      return NUM_result;
    }
```

```
}
```

This code is used in chunk 32.

40. Next we analyze multiplication. If either $\hat{x} = 0$ or $\hat{y} = 0$ then $z = 0$. If $|\hat{z}| < MinDbl$ then $|\hat{x} \cdot \hat{y}| \leq MinDbl$ and hence

$$\begin{aligned} |x \cdot y| &\leq |\hat{x}| \cdot (1 + \epsilon_x) \cdot |\hat{y}| \cdot (1 + \epsilon_y) \\ &\leq MinDbl \cdot (1 + \epsilon_x) \cdot (1 + \epsilon_y) \end{aligned}$$

We set NUM to $MinDbl \cdot (1 + \epsilon_x) \cdot (1 + \epsilon_y)$ and $ERROR$ to $2 * correction$. If $|\hat{z}| \geq MinDbl$ and $|\hat{z}| \neq \infty$ then

$$\begin{aligned} |\hat{z} - (x \cdot y)| &\leq |\hat{z} - \hat{x} \cdot \hat{y}| + |\hat{x} \cdot (\hat{y} - y)| + |y \cdot (\hat{x} - x)| \\ &\leq |\hat{z}| \cdot eps + |\hat{x} \cdot \hat{y}| \cdot \epsilon_y + (|\hat{y}| + |y - \hat{y}|) \cdot |\hat{x}| \cdot \epsilon_x \\ &\leq |\hat{z}| \cdot eps + |\hat{x} \cdot \hat{y}| \cdot \epsilon_y + |\hat{x} \cdot \hat{y}| \cdot (1 + \epsilon_y) \cdot \epsilon_x \\ &= |\hat{z}| \cdot eps + |\hat{x} \cdot \hat{y}| \cdot (\epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y) \end{aligned}$$

Because of $|\hat{x} \cdot \hat{y}| \leq (1 + eps) \cdot |\hat{z}|$, we finally get after dividing both sides by the term $|\hat{z}|$ the estimate

$$\epsilon_{x \cdot y} \leq (\epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y) \cdot (1 + eps) + eps \qquad (2)$$

Since we post-multiply the error bound by the correction factor $1 + 8 \cdot eps$ we may drop the factor $1 + eps$.

⟨ compute error for multiplication 40 ⟩ ≡

```
double &qx = x.PTR→ERROR;
double &qy = y.PTR→ERROR;
if (fabs(NUM) ≥ MinDbl) {
  z_rep.ERROR = qx + qy + qx * qy + eps;
}
else {
  if ((x_num ≠ 0) ∧ (y_num ≠ 0)) {
    NUM = MinDbl * (1 + qx) * (1 + qy);
    z_rep.ERROR = 2;
  }
  else return 0;
}
z_rep.ERROR *= correction;
```

This code is used in chunk 32.

41. Now we come to divisions. The only part where division globally differs from the previous operations is the treatment of the zero–exception. Also, we must set the error to Infinity if the sign of the divisor is unsure.

⟨ Operators 32 ⟩ +≡

```
real operator/(const real &x, const real &y)
{
  ⟨ check for division by zero 42 ⟩
  ⟨ if division is exact return double result 43 ⟩
  ⟨ create nodes for two operands 33 ⟩
```

```
real_rep &z_rep = *new real_rep ( );
⟨ initialize z_rep as result of a binary operation 34 ⟩
z_rep.CON = DIVISION;
NUM = x_num / y_num;
⟨ compute error for division 44 ⟩
⟨ if approximation is not valid set error to Infinity 35 ⟩
return z_rep;
}
```

42. We have to check for division by zero. This is easy to do if the divisor is an exact **double**. (If the sign of the divisor is unsure, we later set the error to infinity).

⟨ check for division by zero 42 ⟩ ≡

```
if (y.is_double) {
  if (y.value ≡ 0) error_handler(1, "real::operator/:Division␣by␣zero");
}
else if ((y.PTR→NUM ≡ 0) ∧ (y.PTR→ERROR < Infinity))
  error_handler(1, "real::operator/:Division␣by␣zero");
```

This code is used in chunk 41.

43. One obvious sufficient condition for the exactness of **double** division is the following. Suppose that both the computed division result *NUM_result* and the dividend *y.NUM* are single–precision. Then the multiplication *NUM_result* ∗ *y.NUM* is exact and we only have to compare with the dividend *x.NUM* to see if *NUM_result* was exact.

⟨ if division is exact return double result 43 ⟩ ≡

```
double NUM_result;
long *num_ptr, *y_ptr;
if (x.is_double ∧ y.is_double) {
  NUM_result = x.value * y.value;
  num_ptr = (long *) &NUM_result;
  y_ptr = (long *) &y.value;
#ifndef LITTLE_ENDIAN
  num_ptr++; y_ptr++;
#endif
  if (¬(*num_ptr & #07ffffff) ∧ ¬(*y_ptr & #07ffffff))
    if (NUM_result * y.value ≡ x.value) return NUM_result;
}
```

This code is used in chunk 41.

44. We analyze the error in **double** division. If $\hat{x} = 0$ then $x = 0$ and hence $z = 0$. If $\hat{y} = 0$ then $y = 0$ and the operation is illegal. If $\epsilon_y \geq 1$ then y may be zero and we can give no approximation for the result. So let us assume $\epsilon_y < 1$, $\hat{x} \neq 0$ and $\hat{y} \neq 0$. If $|\hat{x}/\hat{y}| < MinDbl$ then $|\hat{x}| \oslash |\hat{y} \leq MinDbl$ and

$$|x/y| \leq \frac{|\hat{x}|(1+\epsilon_x)}{|\hat{y}|(1-\epsilon_y)} \leq \frac{MinDbl(1+\epsilon_x)}{1-\epsilon_y}.$$

We use $NUM = MinDbl(1+\epsilon_x)/(1-\epsilon_y)$ and $ERROR = 2 * correction$. Finally, if $|\hat{z}| \geq MinDbl$ then

$$\begin{aligned}
|\widehat{x/y} - x/y| &\leq |\widehat{x/y} - \hat{x}/\hat{y}| + |\hat{x}/\hat{y} - x/y| \\
&\leq |\widehat{x/y}| \cdot eps + |\frac{\hat{x}\cdot y - x \cdot \hat{y}}{y \cdot \hat{y}}| \\
&\leq |\widehat{x/y}| \cdot eps + \frac{|\hat{x}\cdot(\hat{y}-y)| + |\hat{y}\cdot(\hat{x}-x)|}{|y\cdot\hat{y}|} \\
&\leq |\widehat{x/y}| \cdot eps + |\frac{\hat{x}\cdot\hat{y}}{y\cdot\hat{y}}| \cdot (\epsilon_x + \epsilon_y)
\end{aligned}$$

Dividing by $|\widehat{x/y}|$ yields

$$\begin{aligned}
\epsilon_{x/y} &\leq eps + |\hat{y}/y| \cdot (\epsilon_x + \epsilon_y) \cdot (1 + eps) \\
&\leq eps + \frac{|\hat{y}| \cdot (1+eps)}{|\hat{y}| - |\hat{y} - y|} \cdot (\epsilon_x + \epsilon_y) \\
&\leq eps + \frac{\epsilon_x + \epsilon_y}{1 - \epsilon_y} \cdot (1 + eps)
\end{aligned}$$

Again, we may absorb the factor $1 + eps$ into the correction factor.

⟨ compute error for division 44 ⟩ ≡

```
if (y.PTR→ERROR ≥ 1) {
  z_rep.ERROR = Infinity; NUM = 1;
  return z_rep;
}
if (fabs(NUM) ≥ MinDbl)
  z_rep.ERROR = (x.PTR→ERROR + y.PTR→ERROR)/(1 − y.PTR→ERROR) + eps;
else {
  if (x_num ≠ 0) {
    NUM = MinDbl * (1 + x.PTR→ERROR)/(1 − y.PTR→ERROR);
    z_rep.ERROR = 2;
  }
  else return 0;
}
z_rep.ERROR *= correction;
```

This code is used in chunk 41.

45. We come to the squareroot operation. Here we have to take care if the argument is negative or if the sign of the argument is unknown. Otherwise we proceed exactly as before, only we have just one argument.

⟨ Operators 32 ⟩ +≡

```
real sqrt(const real &x)
{
  ⟨ check for negative argument 46 ⟩
  ⟨ if squareroot is exact return double result 47 ⟩
  if (¬x.PTR) ((real &) x).PTR = new real_rep (x.value);
```

```
real_rep &z_rep = *new real_rep ( );
z_rep.OP1 = x.PTR; z_rep.OP1→count++;
z_rep.OP2 = nil;
z_rep.CON = SQUAREROOT;
if (x.PTR→ERROR ≥ 1) {
  z_rep.ERROR = Infinity; z_rep.NUM = 1;
  return real(z_rep);
}
if (x.PTR→NUM < 0)
  error_handler(1, "real::operator␣sqrt:negative␣argument");
z_rep.NUM = sqrt(x.PTR→NUM);
⟨ compute error for squareroot 48 ⟩
return z_rep;
}
```

46. If the argument of squareroot is negative, we exit with an error message.

⟨ check for negative argument 46 ⟩ ≡

```
if ((x.is_double) ∧ (x.value < 0))
  error_handler(1, "real::operator␣sqrt:negative␣argument");
```

This code is used in chunk 45.

47. We give a simple test whether the squareroot is exact. If the squareroot approximation is only "single–precision" and if squaring gives back the original value, then it must be exact. Vice versa, if the approximation has a one after the 26^{th} most significant digit, then it can never give back the original result by squaring.

⟨ if squareroot is exact return double result 47 ⟩ ≡

```
double NUM_result;
long *num_ptr;
if (x.is_double) {
  NUM_result = sqrt(x.value);
  num_ptr = (long *) &NUM_result;
#ifndef LITTLE_ENDIAN
  num_ptr++;
#endif
  if (¬(*num_ptr & #07ffffff)) return NUM_result;
}
```

This code is used in chunk 45.

48. It remains to analyze the error in squareroot.

If $\hat{x} = 0$ then $x = z = 0$. If $\hat{x} \neq 0$ and $\epsilon_x > 1$ or $\hat{x} < 0$ then x may be negative and we can give no meaningful approximation. So let us finally assume that $\hat{x} > 0$ and $\epsilon_x \leq 1$. Then $\hat{x} \geq MinDbl$ and hence $\hat{z} \geq MinDbl$. We have

$$|\widehat{\sqrt{x}} - \sqrt{x}| \quad \leq \quad |\widehat{\sqrt{x}} - \sqrt{\hat{x}}| + |\sqrt{\hat{x}} - \sqrt{x}|$$

$$\begin{aligned}
&\leq \widehat{\sqrt{x}} \cdot eps + \frac{|(\sqrt{\hat{x}} - \sqrt{x}) \cdot (\sqrt{\hat{x}} + \sqrt{x})|}{\sqrt{\hat{x}} + \sqrt{x}} \\
&= \widehat{\sqrt{x}} \cdot eps + \frac{|\hat{x} - x|}{\sqrt{\hat{x}} + \sqrt{x}} \\
&\leq \widehat{\sqrt{x}} \cdot eps + \frac{\hat{x} \cdot \epsilon_x}{\sqrt{\hat{x}}}
\end{aligned}$$

Dividing by $\widehat{\sqrt{x}}$ yields

$$\epsilon_{\sqrt{x}} \leq eps + \epsilon_x \cdot (1 + eps) \tag{3}$$

⟨ compute error for squareroot 48 ⟩ ≡
```
z_rep.ERROR = x.PTR→ERROR + eps;
z_rep.ERROR *= correction;
```
This code is used in chunk 45.

49. Finally we implement the unary "−" operator. Because there is no error in this calculation, we only have to copy the operand with its approximation negated.

⟨ Operators 32 ⟩ +≡
```
real operator−(const real &x)
{
  if (x.is_double) return −x.value;
  real_rep &z_rep = *new real_rep ( );
  z_rep.CON = NEGATION;
  z_rep.OP1 = x.PTR; z_rep.OP1→count++; z_rep.OP2 = nil;
  z_rep.NUM = −x.PTR→NUM; z_rep.ERROR = x.PTR→ERROR;
  return z_rep;
}
```

50. Sign determination

We discuss the different versions of the function **real** :: *sign*. They all compute the sign of the **real** and their implementation rests on the function **real_rep** :: *sign_with_separation_bound*.

⟨ Sign functions 50 ⟩ ≡

```
int real::sign( ) const
{
  if (is_double) return ::sign(value);
  else return PTR->sign_with_separation_bound(zero_integer, false);
}
int real::sign(const integer &q) const
{
  if (is_double) return ::sign(value);
  else return PTR->sign_with_separation_bound(q, true);
}
int real::sign(long p) const
{
  if (is_double) return ::sign(value);
  else return PTR->sign_with_separation_bound(p, true);
}
```

See also chunks 54, 56, 57, 64, and 86.
This code is used in chunk 13.

51. For the implementation of the function *sign_with_separation_bound* we need the concept of a *separation bound*. A separation bound for a real number x is a number q such that $x \leq q$ implies $x = 0$. We allow the user to provide a separation bound. Following Mignotte [Mig92] we compute for each **real** x quantities M and deg such that 2^{-M} is a separation bound for x. The quantities M and deg are defined inductively according to the following rules.

If x is an integer then

$$M(x) = \lceil \log_2(|x|) \rceil, deg(x) = 1,$$

if x is a **bigfloat**, $x = significant \cdot 2^{exponent}$, then we have

$$M = 1 + Max(significant.length() + exponent, -exponent),$$

if $x = y + z$ or $x = y - z$ then

$$\begin{aligned} M(x) &= deg(y) * deg(z) + deg(z) * M(y) + deg(y) * M(z), \\ deg(x) &= deg(y) * deg(z), \end{aligned}$$

if $x = y * z$ or $x = y/z$ then

$$\begin{aligned} M(x) &= deg(z) * M(y) + deg(y) * M(z), \\ deg(x) &= deg(y) * deg(z) \end{aligned}$$

and if $x = \sqrt{y}$ then

$$M(x) = M(y), deg(x) = 2 * deg(y).$$

52. We store the quantities M and *deg* in a structure.

⟨ Basic declarations 16 ⟩ +≡

```
struct sep_bound {
  integer M, deg;
  sep_bound( ) { }
};
```

53. The following function computes the separation bound of the number represented by a node. It asssumes that the separation bounds of the children already exist.

⟨ Local functions of class **real_rep** 53 ⟩ ≡

```
void compute_separation_bound( );
```

See also chunks 55, 67, 69, and 85.
This code is used in chunk 17.

54.⟨ Sign functions 50 ⟩ +≡

```
void real_rep::compute_separation_bound( )
{
  sep_bound *ptr = sep_bound_ptr;
  switch (CON) {
  case DOUBLE: case BIGFLOAT:
    {
      bigfloat &num = APP_BF_PTR→NUM_BF;
      ptr→deg = 1;
      if (isZero(num)) ptr→M = zero_integer;
      else {
        if (sign(num.get_exponent( )) ≥ 0) ptr→M = log2(num);
        else ptr→M = 1 + Maximum(num.get_precision( ), −num.get_exponent( ));
      }
      break;
    }
  case NEGATION: ptr→deg = OP1→sep_bound_ptr→deg;
    ptr→M = OP1→sep_bound_ptr→M;
    break;
  case MULTIPLICATION: case DIVISION: ptr→M = OP1→sep_bound_ptr→M *
        OP2→sep_bound_ptr→deg + OP2→sep_bound_ptr→M * OP1→sep_bound_ptr→deg;
    ptr→deg = OP1→sep_bound_ptr→deg * OP2→sep_bound_ptr→deg;
    break;
  case ADDITION: case SUBTRACTION:
    ptr→deg = OP1→sep_bound_ptr→deg * OP2→sep_bound_ptr→deg;
    ptr→M = ptr→deg + OP1→sep_bound_ptr→M * OP2→sep_bound_ptr→deg +
        OP2→sep_bound_ptr→M * OP1→sep_bound_ptr→deg;
    break;
  case SQUAREROOT: ptr→M = OP1→sep_bound_ptr→M;
    ptr→deg = OP1→sep_bound_ptr→deg ≪ 1;
    break;
```

```
    }
  }
```

55. We also need a recursive function that initializes the separation bound for the whole expression tree of a **real_rep**.

⟨ Local functions of class **real_rep** 53 ⟩ +≡
  void *initialize_separation_bound*( );

56.⟨ Sign functions 50 ⟩ +≡
  void real_rep :: *initialize_separation_bound*( )
  {
    if (*sep_bound_ptr*) **return**;
    if (*OP1*) *OP1*→*initialize_separation_bound*( );
    if (*OP2*) *OP2*→*initialize_separation_bound*( );
    sep_bound_ptr = **new sep_bound** ( );
    compute_separation_bound( );
  }

57. Now we come to the function *sign_with_separation_bound*, the core of this section. The basic structure of the function is as follows. We first decide whether the current approximations already determine the sign. If so, we return the sign immediately. Otherwise, we first compute a separation bound for the **real_rep**. In the main part of *sign*, we repeatedly improve the quality of the **bigfloat** approximation. As long as the current precision is less than q and less than the separation bound and the current approximation does not determine the sign we square the precision and repeat. If after that loop the sign is still not clear from the current approximation, the sign must be zero and we can remove the operand pointers.

⟨ Sign functions 50 ⟩ +≡
  int real_rep :: *sign_with_separation_bound*(**const integer** &q, **bool** *supported*)
  {
    ⟨ if current approximations determine the sign return it 58 ⟩
    initialize_separation_bound( );
    ⟨ compute bigfloat approximations of increasing precision 60 ⟩
    if ⟨ sign is clear from the bigfloat approximation 59 ⟩ **return** *sign*(*num_bf*( ));
    ⟨ set number to zero 61 ⟩
    return 0;
  }

58. When do the current approximations determine the sign? The **double** approximations does when either $NUM \equiv 0$ or $ERROR < 1$.

⟨ if current approximations determine the sign return it 58 ⟩ ≡
  if $((ERROR \neq Infinity) \wedge ((ERROR < 1) \vee (NUM \equiv 0)))$ **return** :: *sign*(*NUM*);
  init_app_bf( );
  if ⟨ sign is clear from the bigfloat approximation 59 ⟩ **return** *sign*(*num_bf*( ));

This code is used in chunk 57.

59. The **bigfloat** approximation determines the sign when $ERROR\_BF \equiv 0$ or $ERROR\_BF < abs(NUM\_BF)$.

$\langle$ sign is clear from the bigfloat approximation 59 $\rangle \equiv$
$(exact() \lor (\neg isZero(num\_bf()) \land (num\_exp() > err\_exp())))$

This code is used in chunks 57, 58, and 60.

60. Our current approximations do not determine the sign and we have computed a maximal precision bound q. Now we repeatedly improve the working precision until either precision q or the bound *bound* is reached or the sign is determined.

$\langle$ compute bigfloat approximations of increasing precision 60 $\rangle \equiv$

```
bool user_bound_is_reached = false;
integer relative_precision = 26;
integer absolute_precision;
do {
   relative_precision = relative_precision * 2;
   absolute_precision = err_exp() − relative_precision;
   guarantee_bound_two_to(absolute_precision);
   user_bound_is_reached = supported ∧ (absolute_precision ≥ q);
} while (¬(user_bound_is_reached ∨ ⟨ sign is clear from the bigfloat approximation 59 ⟩));
```

This code is used in chunk 57.

61. Finally we assume that the number is smaller than the computed separation bound and consequently is zero.

$\langle$ set number to zero 61 $\rangle \equiv$

```
NUM = ERROR = 0;
num_bf() = error_bf() = pZero;
⟨ remove operand pointers 62 ⟩
```

This code is used in chunks 57 and 66.

62. $\langle$ remove operand pointers 62 $\rangle \equiv$

```
{
   if (OP1) if (−−OP1→count ≡ 0) delete OP1;
   if (OP2) if (−−OP2→count ≡ 0) delete OP2;
   OP1 = OP2 = nil;
   CON = BIGFLOAT;
}
```

This code is used in chunks 61, 66, and 68.

63. The comparison operators could have been directly derived from the sign function in the obvious way. But then in most cases we would have to allocate an obsolete new node by calling the subtraction operator. This can be avoided if we use a straightforward test that tells us if the comparison is easy and can be done by comparing the **double**s.

```
⟨ check if comparison is easy 63 ⟩ ≡
  double x_error, y_error;
  bool is_easy;
  if (x.is_double ∧ y.is_double) is_easy = true;
  else {
    if (x.is_double) x_error = 0;
    else {
      ((real &) x).value = x.PTR→NUM;
      x_error = fabs(x.value) * x.PTR→ERROR;
    }
    if (y.is_double) y_error = 0;
    else {
      ((real &) y).value = y.PTR→NUM;
      y_error = fabs(y.value) * y.PTR→ERROR;
    }
    is_easy = (fabs(x.value − y.value) > (x_error + y_error + twoMinDbl) * correction);
  }
```

This code is used in chunk 64.

64. Now we can give the comparison operators. If we know that the comparison is easy, then we compare the **double** values. Otherwise we call the sign function.

```
⟨ Sign functions 50 ⟩ +≡
  int operator≡(const real &x, const real &y)
  {
    ⟨ check if comparison is easy 63 ⟩
    if (is_easy) return (x.value ≡ y.value);
    return ((y − x).sign() ≡ 0);
  }
  int operator≠(const real &x, const real &y)
  {
    ⟨ check if comparison is easy 63 ⟩;
    if (is_easy) return (x.value ≠ y.value);
    return ((y − x).sign() ≠ 0);
  }
  int operator < (const real &x, const real &y)
  {
    ⟨ check if comparison is easy 63 ⟩;
    if (is_easy) return (x.value < y.value);
    return ((y − x).sign() > 0);
  }
  int operator > (const real &x, const real &y)
  {
    ⟨ check if comparison is easy 63 ⟩;
    if (is_easy) return (x.value > y.value);
    return ((y − x).sign() < 0);
  }
```

int operator$\leq$(**const real** $\&x$, **const real** $\&y$)
{
  $\langle$ check if comparison is easy 63 $\rangle$;
  if ($\mathit{is\_easy}$) **return** ($x.\mathit{value} \leq y.\mathit{value}$);
  return ($(y - x).\mathit{sign}() \geq 0$);
}
int operator$\geq$(**const real** $\&x$, **const real** $\&y$)
{
  $\langle$ check if comparison is easy 63 $\rangle$;
  if ($\mathit{is\_easy}$) **return** ($x.\mathit{value} \geq y.\mathit{value}$);
  return ($(y - x).\mathit{sign}() \leq 0$);
}

65. The functions improve and related functions

Function **real** :: *improve_approximation_to* is realized by calling function **real_rep** :: *guarantee_bound_two_to* with the sign of the argument inversed. The reason to invert the sign is that this syntax makes it easier to understand the numerical details of *guarantee_bound_two_to*.

⟨ Functions for improving the approximation 65 ⟩ ≡

```
void real::improve_approximation_to(const integer &p) const
{
  if (is_double) return;
  PTR→init_app_bf( );
  PTR→guarantee_bound_two_to(−p);
  adjust( );
}
```

See also chunks 66, 68, 72, 76, 77, 78, 79, 80, 81, 82, 84, and 87.

This code is used in chunk 13.

66. The function *guarantee_bound_two_to* is among the core of our data structure. The call *guarantee_bound_two_to*(**integer** e) improves *error_bf*( ) to a value $\le 2^e$. For each of the operations addition, subtraction, multiplication, division and squareroot, we have a function *improve_operation* to do this job. In case that the **bigfloat** approximation is exact, we remove the obsolete pointers *OP1* and *OP2*. If the function changes the **bigfloat** approximation of the **real_rep** it also updates its separation bound. After doing so we check whether the new separation bound and the new approximation together imply that the number is zero.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real_rep::guarantee_bound_two_to(const integer &e)
{
  bool current_precision_is_sufficient = isZero(error_bf( )) ∨ (err_exp( ) ≤ e);
  if (current_precision_is_sufficient) return;
  switch (CON) {
  case DOUBLE: case BIGFLOAT: return;
  case NEGATION: OP1→guarantee_bound_two_to(e);
    num_bf( ) = −OP1→num_bf( );
    error_bf( ) = OP1→error_bf( );
    break;
  case ADDITION: improve_add(e); break;
  case SUBTRACTION: improve_sub(e); break;
  case MULTIPLICATION: improve_mul(e); break;
  case DIVISION: improve_div(e); break;
  case SQUAREROOT: improve_sqrt(e); break;
  }
  if (exact( )) ⟨ remove operand pointers 62 ⟩
  if (sep_bound_ptr) {
    compute_separation_bound( );
    if (log2(error_bf( ) + abs(num_bf( ))) ≤ −sep_bound_ptr→M) {
```

```
      ⟨ set number to zero 61 ⟩
      sep_bound_ptr→M = zero_integer;
      sep_bound_ptr→deg = 1;
    }
  }
  adjust_dbl();
}
```

67. At this moment we list only the prototypes of the functions *improve_op*. The function bodies will be given later, when we present the functions *compute_op*.

```
⟨ Local functions of class real_rep 53 ⟩ +≡
  void improve_add(const integer &);
  void improve_sub(const integer &);
  void improve_mul(const integer &);
  void improve_div(const integer &);
  void improve_sqrt(const integer &);
```

68. We implement the function **real_rep** :: *compute_op* that we used in **real** :: *init_app_bf*. The basic structure of *compute_op* is as follows. Suppose that the **real_rep** object was constructed by the operation *op* and has operands that represent numbers x and y with approximations $\hat{x}$ and $\hat{y}$. Again we distinguish the respective cases of *op*. First we compute $\hat{x}$ *op* $\hat{y}$ in precision p. The error in this operation alone is called the *operation–induced* error and is given by $|\hat{x}\ op\ \hat{y}| \cdot 2^{-p}$. Then we compute the error arising from the inaccuracies of $\hat{x}$ and $\hat{y}$, which we call the *operand–induced* error. For multiplication, division and squareroot we use special functions to compute the operand–induced error. The overall error of the approximation finally is the sum of the operand–induced and the operation–induced error. We remark that for division the sign of the dividend must be known to be nonzero and similarly the argument of squareroot must known to be nonnegative.

```
⟨ Functions for improving the approximation 65 ⟩ +≡
  void real_rep::compute_op(long p)
  {
    bool isexact;
    bigfloat operation_error, operand_error;
    switch (CON) {
    case DOUBLE: case BIGFLOAT: return;
    case NEGATION: num_bf() = −OP1→num_bf();
      error_bf() = OP1→error_bf();
      break;
    case ADDITION:
      num_bf() = add(OP1→num_bf(), OP2→num_bf(), p, TO_NEAREST, isexact);
      ⟨ compute operand error for addition and subtraction 71 ⟩
      break;
    case SUBTRACTION:
      num_bf() = sub(OP1→num_bf(), OP2→num_bf(), p, TO_NEAREST, isexact);
```

$\langle$ compute operand error for addition and subtraction 71 $\rangle$
break;
case *MULTIPLICATION*:
$num\_bf() = mul(OP1 \rightarrow num\_bf(), OP2 \rightarrow num\_bf(), p, TO\_NEAREST, isexact)$;
$compute\_mul\_error(operand\_error)$;
break;
case *DIVISION*:
if $(OP2 \rightarrow sign\_with\_separation\_bound(zero\_integer, false) \equiv 0)$
$error\_handler(1,$ `"compute_op:division␣by␣zero"`$)$;
$num\_bf() = div(OP1 \rightarrow num\_bf(), OP2 \rightarrow num\_bf(), p, TO\_NEAREST, isexact)$;
$compute\_div\_error(operand\_error)$;
break;
case *SQUAREROOT*:
if $(OP1 \rightarrow sign\_with\_separation\_bound(zero\_integer, false) < 0)$
$error\_handler(1,$ `"compute_op:sqrt␣of␣negative␣number"`$)$;
$num\_bf() = sqrt(OP1 \rightarrow num\_bf(), p, TO\_NEAREST, isexact)$;
$compute\_sqrt\_error(operand\_error)$;
}
if $(\neg isexact)$ $operation\_error = ldexp\_bf(num\_bf(), -p)$;
$error\_bf() = operation\_error + operand\_error$;
if $(exact())$ $\langle$ remove operand pointers 62 $\rangle$
$adjust\_dbl()$;
}

69. We list the prototypes of the functions *compute_op_error*.

$\langle$ Local functions of class **real_rep** 53 $\rangle$ $+\equiv$
void *compute_mul_error*(**bigfloat** &);
void *compute_div_error*(**bigfloat** &);
void *compute_sqrt_error*(**bigfloat** &);

70. We now give the details of *compute_op_error* and *improve_op* for the various cases of numerical operations *op*. In the context of *compute_op_error* we want to compute the error q of an approximation $\hat{z}$ of some real number z. For the function *improve*, we face the opposite problem. There we are *given* a bound q and we want to determine an approximation $\hat{z}$ with error at most q. In both cases, the number z is defined by two (one in the case of squareroot) other numbers x and y by means of the operation *op*.

71. Let's deal with addition and subtraction first. We have

$$\begin{aligned} |\hat{z} - z| &\leq |\hat{z} - (\hat{x} + \hat{y})| + |\hat{x} - x| + |\hat{y} - y| \\ &\leq \epsilon \cdot |\hat{z}| + qx + qy \end{aligned} \tag{4}$$

where ϵ is the relative precision of the **bigfloat** addition. We use this formula to compute the error bounds for addition and subtraction directly with **bigfloat**s.

$\langle$ compute operand error for addition and subtraction 71 $\rangle$ $\equiv$
$operand\_error = add(OP1 \rightarrow error\_bf(), OP2 \rightarrow error\_bf(), 16, TO\_INF)$;

This code is used in chunk 68.

72. Let us discuss how to improve the operations addition and subtraction. Here we need to determine two bounds qx and qy and a relative precision ϵ such that the following is true: If $\hat{x}$ and $\hat{y}$ are approximations of x and y of error qx and qy respectively and $\hat{z}$ is computed from $\hat{x}$ and $\hat{y}$ by performing operation *op* with relative precision ϵ then $\hat{z}$ has maximal error q. Looking again at Inequality 4, we are done if we bound the first term (the operation induced error) by $q/2$ and the second and third term (the operand errors) by $q/4$. That is, we need

$$\epsilon \leq \frac{q}{2|\hat{z}|}, qx \leq q/4, qy \leq q/4.$$

The second two conditions are easy to satisfy. Taking $q = 2^e$, we only need to call $x.guarantee\_bound\_two\_to(e-2)$ and $y.guarantee\_bound\_two\_to(e-2)$. How can we guarantee the first condition? The problem is that we do not know $\hat{z}$. So we replace $\log(\hat{z})$ by the crude bound $Max(\log(\hat{x}), \log(\hat{y})) + 1$.

⟨ Functions for improving the approximation 65 ⟩ +≡
  void real_rep :: *improve_add*(**const integer** &*e*)
  {
    long *p*;
    ⟨ improve operands and compute p for add and sub 73 ⟩
    bool *isexact*;
    num_bf( ) = *add*(*OP1*→*num_bf*( ), *OP2*→*num_bf*( ), *p*, *TO_NEAREST*, *isexact*);
    ⟨ set error 2^e 75 ⟩
  }
  void real_rep :: *improve_sub*(**const integer** &*e*)
  {
    long *p*;
    ⟨ improve operands and compute p for add and sub 73 ⟩
    bool *isexact*;
    num_bf( ) = *sub*(*OP1*→*num_bf*( ), *OP2*→*num_bf*( ), *p*, *TO_NEAREST*, *isexact*);
    ⟨ set error 2^e 75 ⟩
  }

73.⟨ improve operands and compute p for add and sub 73 ⟩ ≡
  OP1→*guarantee_bound_two_to*(*e* − 2);
  OP2→*guarantee_bound_two_to*(*e* − 2);
  integer *log_epsilon* = −1;
  if (¬*isZero*(*OP1*→*num_bf*( )) ∧ ¬*isZero*(*OP2*→*num_bf*( )))
    log_epsilon = *e* − *Maximum*(*OP1*→*num_exp*( ), *OP2*→*num_exp*( )) − 2;
  ⟨ convert log_epsilon to p 74 ⟩

This code is used in chunk 72.

74.⟨ convert log_epsilon to p 74 ⟩ ≡
  if (¬*log_epsilon*.*islong*( )) {
    error_handler(1, "sorry:improve:computation␣to␣expensive");
    p = 10000;   /* just to silence compiler warnings */

```
}
else p = −log_epsilon.tolong( );
```

This code is used in chunks 73, 77, 79, and 81.

75.⟨ set error 2^e 75 ⟩ ≡

```
if (isexact ∧ OP1→exact( ) ∧ OP2→exact( )) error_bf( ) = pZero;
else error_bf( ) = pow2(e);
```

This code is used in chunks 72, 77, and 79.

76. We deal with multiplication next. We have

$$\begin{aligned} |\widehat{x \cdot y} - (x \cdot y)| &\le |\widehat{x \cdot y} - (\hat{x} \cdot \hat{y})| + |\hat{x} \cdot (\hat{y} - y)| + |y \cdot (\hat{x} - x)| \\ &\le |\widehat{x \cdot y}| \cdot \epsilon + |\hat{x}| \cdot qy + |y| \cdot qx \end{aligned} \tag{5}$$

We first implement the part of computing errors. Since multiplications are expensive for operands with large precision, we first round them to 16 bit and then do the multiplications in precision 32. By choosing rounding mode "to ∞" we avoid having to care for the inaccuracies.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real_rep::compute_mul_error(bigfloat &operand_error)
{
  bigfloat x = OP1→num_bf( );
  bigfloat y = OP2→num_bf( );
  bigfloat error_x = mul(abs(round(x, 16, TO_INF)), OP2→error_bf( ), 32, TO_INF);
  bigfloat error_y = mul(abs(round(y, 16, TO_INF)), OP1→error_bf( ), 32, TO_INF);
  operand_error = add(error_x, error_y, 16, TO_INF);
}
```

77. Now we implement the improve function for multiplication. Again, our goal is to bound the first term of Relation 5 by $q/2$ and the two other terms by $q/4$, or equivalently

$$\epsilon \le \frac{q}{2|\widehat{x \cdot y}|}, qx \le \frac{q}{4|y|}, qy \le \frac{q}{4|\hat{x}|} \tag{6}$$

Since we don't know y, we bound it by $y\_high = abs(y.num\_bf()) + y.error\_bf()$.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real_rep::improve_mul(const integer &e)
{
  bigfloat y_high = add(abs(OP2→num_bf( )), OP2→error_bf( ), 32, TO_P_INF);
  integer ex = e − log2(y_high) − 2;
  OP1→guarantee_bound_two_to(ex);
  integer log_epsilon = −1;
  if (¬isZero(OP1→num_bf( ))) {
    integer ey = e − OP1→num_exp( ) − 2;
```

```
    OP2→guarantee_bound_two_to(ey);
    log_epsilon = e − (OP1→num_exp() + OP2→num_exp()) − 1;
  }
  bool isexact;
  long p;
  ⟨ convert log_epsilon to p 74 ⟩
  num_bf() = mul(OP1→num_bf(), OP2→num_bf(), p, TO_NEAREST, isexact);
  ⟨ set error 2^e 75 ⟩
}
```

78. Division is next. Remember that when the functions *compute_div_error* and *improve_div* are called the sign of the divisor y is known to be nonzero, because this exception was handled in the division operator or by *init_app_bf*. Hence we can use $y\_low = abs(y.num\_bf()) - y.error\_bf() > 0$ in our calculation. Let us consider the error estimate given for the division operator.

$$\begin{aligned} \left|\frac{\widehat{x}}{y} - \frac{x}{y}\right| &\leq \left|\frac{\widehat{x}}{y}\right| \cdot \epsilon + \frac{|\hat{x}| \cdot qy}{|y \cdot \hat{y}|} + \frac{qx}{|y|} \\ &= \left|\frac{\widehat{x}}{y}\right| \cdot \epsilon + \frac{1}{|y|} \cdot \left(\frac{|\hat{x}| \cdot qy}{|\hat{y}|} + qx\right) \end{aligned} \tag{7}$$

As before we avoid multiplication with operands having large precisons and also we avoid divisions completely.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
  void real_rep::compute_div_error(bigfloat &operand_error)
  {
    bigfloat y_low = sub(abs(OP2→num_bf()), OP2→error_bf(), 32, TO_N_INF);
    bigfloat x = OP1→num_bf();
    bigfloat z = mul(abs(round(x, 16)), OP2→error_bf(), 32, TO_INF);
    z = ldexp_bf(z, 1 − OP2→num_exp());
    operand_error = add(z, abs(OP1→error_bf()), 16, TO_INF);
    operand_error = ldexp_bf(operand_error, 1 − log2(y_low));
  }
```

79. In the div version of improve, we again want to bound the last two terms of Relation 7 by $q/4$ and the first one by $q/2$. This means

$$\epsilon \leq \frac{q}{2|\hat{z}|}, qx \leq \frac{q \cdot y}{4}, qy \leq \frac{q \cdot y \cdot |\hat{y}|}{4|\hat{x}|} \tag{8}$$

We proceed exactly as for the other cases of *improve*.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
  void real_rep::improve_div(const integer &e)
  {
    bigfloat y_low = sub(abs(OP2→num_bf()), OP2→error_bf(), 32, TO_N_INF);
    integer ex = e + log2(y_low) − 3;
```

```
    OP1->guarantee_bound_two_to(ex);
    integer log_epsilon = -1;
    if (¬isZero(OP1->num_bf())) {
      integer ey = ex + OP2->num_exp() - OP1->num_exp() - 3;
      OP2->guarantee_bound_two_to(ey);
      log_epsilon = e + OP2->num_exp() - OP1->num_exp() - 2;
    }
    bool isexact;
    long p;
    ⟨convert log_epsilon to p 74⟩
    num_bf() = div(OP1->num_bf(), OP2->num_bf(), p, TO_NEAREST, isexact);
    ⟨set error 2^e 75⟩
  }
```

80. We finally deal with the squareroot operation. We have

$$|\widehat{\sqrt{x}} - \sqrt{x}| \leq \widehat{\sqrt{x}} \cdot \epsilon + \frac{|\hat{x} - x|}{\sqrt{\hat{x}} + \sqrt{x}} \leq \widehat{\sqrt{x}} \cdot \epsilon + \frac{qx}{\sqrt{\hat{x}}} \tag{9}$$

We bound the logarithm of $\sqrt{\hat{x}}$ by $(x.num\_exp() + 1)/2$.

⟨Functions for improving the approximation 65⟩ +≡

```
  void real_rep::compute_sqrt_error(bigfloat &operand_error)
  {
    integer log_sqrt = (num_exp() + 1) >> 1;
    if (¬OP1->exact()) operand_error = ldexp_bf(OP1->error_bf(), -log_sqrt);
  }
```

81. Last we show how to improve the squareroot operation. To bound the two terms on the right hand side of Estimate 9 by $q/2$, we set $qx = q \cdot \sqrt{x}/2$ and $\epsilon = q/(2\widehat{\sqrt{x}})$. Since we don't know $\sqrt{x}$, we bound its binary logarithm by $\lfloor \log_2(x\_low)/2 \rfloor$ from below and by $\lceil \log_2(x\_high)/2 \rceil$ from above.

⟨Functions for improving the approximation 65⟩ +≡

```
  void real_rep::improve_sqrt(const integer &e)
  {
    bool isexact;
    bigfloat x_low = sub(OP1->num_bf(), OP1->error_bf(), 32, TO_N_INF);
    integer ex = e + (log2(x_low) - 1)/2 - 1;
    OP1->guarantee_bound_two_to(ex);
    bigfloat x_high = add(OP1->num_bf(), OP1->error_bf(), 32, TO_P_INF);
    integer log_epsilon = e - (log2(x_high) + 1)/2 - 1;
    long p;
    ⟨convert log_epsilon to p 74⟩
    num_bf() = sqrt(OP1->num_bf(), p, TO_NEAREST, isexact, num_bf());
    if (isexact ∧ OP1->exact()) error_bf() = pZero;
    else error_bf() = pow2(e);
  }
```

82. It remains to implement the functions *x.guarantee_relative_error* and *x.compute_with_precision*(p). We begin with function *guarantee_relative_error*. It first performs a sign test. If the sign is nonzero, it calls *improve* with an appropriate error bound.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real::guarantee_relative_error(long p) const
{
  if (is_double) return;
  if (PTR→sign_with_separation_bound(zero_integer) ≡ 0) return;
  PTR→init_app_bf();
  bigfloat num_low = abs(PTR→num_bf()) − PTR→error_bf();
  PTR→guarantee_bound_two_to(log2(num_low) − p);
  adjust();
}
```

83. Function *compute_with_precision* uses the function *compute_approximation* that in turn calls *compute_op* for every unvisited node in the expression tree. To allow intelligent searching in the expression tree, we assign values of an enumeration type *statustype* to the member *status*. The meaning of these states is the following.

- *EMPTY* is the initial state.
- *VISITED* means that the computation for the node has been successfully finished.
- *CLEARED* is used by the function *clear_visited_marks* that we define in a moment.

⟨ Basic declarations 16 ⟩ +≡

```
enum statustype { EMPTY = 0, CLEARED = 1, VISITED = 2 };
```

84.⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real::compute_with_precision(long p) const
{
  if (is_double) return;
  PTR→init_app_bf();
  PTR→clear_visited_marks();
  PTR→compute_approximation(p);
  adjust();
}
```

85. We need a function to clear the visited marks used by *compute_approximation*.

⟨ Local functions of class **real_rep** 53 ⟩ +≡

```
void clear_visited_marks();
```

86. This function is implemented as follows. We maintain the invariant that if any node is in the state *CLEARED*, then all children of that node are *CLEARED* too.

⟨ Sign functions 50 ⟩ +≡

```
void real_rep :: clear_visited_marks( )
{
  if (status ≡ CLEARED) return;
  if (OP1) OP1→clear_visited_marks( );
  if (OP2) OP2→clear_visited_marks( );
  status = CLEARED;
}
```

87. Now we consider the function *compute_approximation*(**long** p). When the function is called for the object *x_rep*, we compute every operation in the expression tree of *x_rep* with **bigfloat**s in precision p. The final result is stored as the new **bigfloat** approximation of *x_rep*.

⟨ Functions for improving the approximation 65 ⟩ +≡

```
void real_rep :: compute_approximation(long p)
{
  if (status ≡ VISITED) return;
  if (OP1) OP1→compute_approximation(p);
  if (OP2) OP2→compute_approximation(p);
  compute_op(p);
  status = VISITED;
}
```

88. References.

References

[BJMM93] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A "lazy" solution to imprecision in computational geometry. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 73–78, 1993.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *LNCS*, volume 855, pages 227–239. Springer-Verlag Berlin/New York, 1994. Proceedings of ESA'94.

[FM91] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proceedings of the 7th ACM Symposium on Computational Geometry*, pages 334–341, 1991.

[For92] S. Fortune. Numerical stability of algorithms for 2d Delaunay triangulations and Voronoi diagrams. In *Proceedings of the 8th Symposium on Computational Geometry*, pages 83–92, 1992.

[Fv93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. *Proceedings of the 9th Symposium on Computational Geometry*, pages 163–171, 1993.

[IEE87] IEEE standard 754-1985 for binary floating–point arithmetic. *reprinted in SIGPLAN 22*, 2:9–25, 1987.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[Knu80] D.E. Knuth. *The Art of Computer Programming, Volume II.* Addison-Wesley, 1980.

[LM92] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.

[Mig92] M. Mignotte. *Mathematics for Computer Algebra.* Springer Verlag, 1992.

[MN94a] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[MN94b] K. Mehlhorn and St. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[MN95] K. Mehlhorn and St. Näher. LEDA: A library of efficient data types and algorithms. *CACM*, 38(1):96–102, 1995.

[Näh95] St. Näher. LEDA Manual Version 3.2. Technical Report MPI-I-95-1-002, Max-Planck-Institut für Informatik, 1995.

[Yap93] C.K. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 405–419, 1993.

[Yap94] C.K. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1994.

[YD95] C. Yap and T. Dube. The exact computation paradigm. In *Computing in Euclidean Geometry II.* World Scientific Press, 1995.

Index

List of Refinements