High-Precision Floating Point Numbers in LEDA *

Christoph Burnikel   Jochen Könemann

January 26, 1996

Contents

*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).

1. Introduction

The data type *bigfloat* is the high-precision floating point type of LEDA. A bigfloat is a number of the form $s \cdot 2^e$ where s and e are integers. s is called the *significant* or *mantissa* and e is called the *exponent*. Arithmetic on bigfloats is governed by two parameters : the mantissa length and the rounding mode. Both parameters can either be set globally or for a single operation. The arithmetic on bigfloats works as follows: first the exact result of an operation is computed and then the mantissa is rounded to the prescribed number of digits as dictated by the rounding mode. The available rounding modes are *TO_NEAREST* (round to the nearest representable number), *TO_P_INF* (round upwards), *TO_N_INF* (round downwards), *TO_ZERO* (round towards zero), *TO_INF* (round away from zero) and *EXACT*. The latter mode only applies to addition, subtraction and multiplication. In this mode the *precision* parameter is ignored and no rounding takes place. Since the exponents of bigfloats are arbitrary integers (type integer) arithmetic operations never underflow or overflow. However, exceptions (division by zero, square root of a negative number) may occur. They are handled according to the IEEE floating point standard, e.g. $5/0$ evaluates to ∞, $-5/0$ evaluates to $-\infty$, $+\infty + 5$ evaluates to $+\infty$ and $0/0$ evaluates to *NaN* (=not a number). This report is structured as follows. Section 2 defines the bigfloat through its manual page and the remaining sections contain the implementation. The implementation is split into files *bigfloat.h* and *bigfloat.c*.

2. The Manual Page of data type bigfloat

1. Definition

In general a *bigfloat* is given by two integers s and e where s is the significant and e is the exponent. The tuple (s, e) represents the real number

$$s \cdot 2^e.$$

In addition, a bigfloat can be in a special state like *NAN* (= not a number), *PZERO*, *NZERO* ($= +0, -0$), and *PINF*, *NINF* ($= +\infty, -\infty$). bigfloats in special states behave as defined by the IEEE floating point standard. In particular, $\frac{5}{+0} = \infty$, $\frac{-5}{+0} = -\infty$, $\infty + 1 = \infty$, $\frac{5}{\infty} = +0$, $+\infty + (-\infty) = NaN$ and $0 \cdot \infty = NaN$ (NaN is a bigfloat in special state *NAN*). Arithmetic on bigfloats uses two parameters: The precision *prec* of the result (in number of binary digits) and the rounding mode *mode*. Possible rounding modes are:

- *TO_NEAREST*: round to the closest representable value
- *TO_ZERO*: round towards zero
- *TO_INF*: round away from zero
- *TO_P_INF*: round towards $+\infty$
- *TO_N_INF*: round towards $-\infty$
- *EXACT*: do not round at all for $+, -, *$ and round to nearest otherwise

Operations $+$, $-$, $*$ work as follows. First, the exact result z is computed. If the rounding mode is EXACT then z is the result of the operation. Otherwise, let s be the significant of the result; s is rounded to *prec* binary places as dictated by *mode*. Operations / and $\sqrt{}$ work accordingly except that EXACT is treated as TO_NEAREST.

The parameters *prec* and *mode* are either set directly for a single operation or else they are set globally for every operation to follow. The default values are 53 for *prec* and TO_NEAREST for *mode*.

2. Creation

A bigfloat may be constructed from data types double, long, int and integer, without loss of accuracy. In addition, an instance of type bigfloat can be created as follows.

bigfloat x(*integer* s, *integer* e);

introduces a variable x of type bigfloat and initializes it $s \cdot 2^e$

bigfloat x(*special_values sp*);

creates an instance of a special value of type bigfloat

3. Operations

The arithmetical operators $+$, $-$, $*$, $/$, $sqrt$, the comparison operators $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ and the stream operators $\ll$ and $\gg$ are available.

Addition, subtraction, multiplication, division and square root are implemented by the functions *add*, *sub*, *mul*, *div* and *sqrt*, respectively. For example, the call

$$add(x, y, prec, mode, is\_exact)$$

computes the sum of bigfloats x and y with *prec* binary digits, in rounding mode *mode*, and returns it. The optional last parameter *is_exact* is a boolean variable that is set to *true* if and only if the returned bigfloat exactly equals the sum of x and y.

The parameters *prec* and *mode* are also optional and have the global default values *global_prec* and *round_mode* respectively, that is, the three calls $add(x, y, glob\_prec, round\_mode)$, $add(x, y, glob\_prec)$, and $add(x, y)$ are all equivalent. The syntax for functions *sub*, *mul*, *div*, and *sqrt* is analogous.

The operators $+$, $-$, $*$, and $/$ are implemented by their counterparts among the functions *add*, *sub*, *mul* and *div*. For example, the call $x + y$ is equivalent to $add(x, y)$.

The rounded value of a bigfloat x can be obtained by

$$round(x, prec, mode, is\_exact, bias)$$

Here *bias* is an optional long variable taking values in $\{-1, 0, +1\}$, with default value 0. The function *round* rounds $x + bias \cdot \epsilon$, where ϵ is a positive infinitesimal, with prec binary digits in rounding mode *mode*. The optional boolean variable *is_exact* is set to true iff the rounding operation did not change the value of x and $bias == 0$.

bigfloats offer a small set of mathematical functions (e.g. *abs*, *log2*, *ceil*, *floor*, *sign*), functions to test for special values, conversions to doubles and integers, functions to access *significant* and *exponent*, and functions to set the global precision, the rounding mode and the output mode.

bool isNaN(*bigfloat x*) returns *true* iff x is in special state *NAN*

bool isnInf(*bigfloat x*) returns *true* iff x is in special state *NINF*

bool ispInf(*bigfloat x*) returns *true* iff x is in special state *PINF*

bool isnZero(*bigfloat x*)
returns *true* iff x is in special state *NZERO*

bool ispZero(*bigfloat x*)
returns *true* iff x is in special state *PZERO*

bool isZero(*bigfloat* x) returns *true* iff *ispZero*(x) or *isnZero*(x)

bool isInf(*bigfloat* x) returns *true* iff *ispInf*(x) or *isnInf*(x)

bool isSpecial(*bigfloat* x)
returns *true* iff x is in a special state

long sign(*bigfloat* x) computes the sign of x.

long sign_of_special_value(*bigfloat* x)
computes the sign of special values. For example:
sign_of_special_value (bigfloat(PZERO)) == 1

bigfloat abs(*bigfloat* x) the absolute value of x

bigfloat pow2(*integer* p) returns 2^p

integer log2(*bigfloat* x) returns the binary logarithm of x, rounded up to the next integer. *Precondition*: $x > 0$

integer ceil(*bigfloat* x) rounds x up to the next integer

integer floor(*bigfloat* x) rounds x down to the next integer

double todouble(*bigfloat* x)
returns the double value next to x (in rounding mode *TO_NEAREST*

integer tointeger(*bigfloat* x, *rounding_modes rmode = TO_ZERO*)
returns the integer value next to x (in the given rounding mode)

ostream& ostream& os $\ll$ x
writes x to output stream *os*

istream& istream& is $\gg$ *bigfloat&* x
reads x from input stream *is*

long x.get_precision(*void*)
returns the precision, i.e. the length of the significant of x.

integer x.get_exponent(*void*)
returns the exponent of x

integer x.get_significant(*void*)
returns the significant of x.

void $\quad$ *bigfloat*::set_glob_prec(*long p*)

sets the global precision value to p

void $\quad$ *bigfloat*::set_round_mode(*rounding_modes m = TO_NEAREST*)

sets the global rounding mode

void $\quad$ *bigfloat*::set_output_mode(*output_modes o_mode = DEC_OUT*)

sets the output mode

3. Representation of bigfloats

A bigfloat is stored as four quantities: integers *significant* and *exponent*, a flag *special* of enumeration type *specialvalues* with elements NOT, $PZERO$, $NZERO$, $PINF$, $NINF$, and NAN, and a long *precision*. We maintain the following invariants:

1. if $special \equiv NOT$ then the number represented by the bigfloat is $significant \cdot 2^{exponent}$, and *precision* is the number of binary digits in the *significant*

2. if $special \neq NOT$ then *special* is the value of the bigfloat

4. A bigfloat number does not necessarily have a unique representation because there may be zeros at the end of the *significant*. However, some functions require a unique representation. We call a bigfloat *normalized* if its *significant* ends in a one. To guarantee uniqueness some functions normalize their input before working on it. To do this we have an internal procedure *normalize*. Warning: The value 0 can have many different representations. After calling the function *normalize* a bigfloat is special with value $PZERO$ or $NZERO$ iff its value is 0.

$\langle$ functions for internal use 4 $\rangle \equiv$

```
void bigfloat::normalize(void)
{
  if (special ≠ NOT) return;
  int signum = ::sign(significant);
  if ((signum ≡ 0) ∧ (special ≡ NOT)) special = PZERO;
  long z = significant.zeros( );
      /* z is the number of final zeros in the significant */
  if (z > 0) significant = significant ≫ z;
  precision = significant.length( );
  exponent += z;
}
```

See also chunks 19, 54, 57, 75, 76, 77, and 84.

This code is used in chunk 10.

5. The Header-File

The header file contains the prototypes of all the functions mentioned in the manual and some additional material: the definition of the rounding modes, the output modes, and the special values, the definition of some private functions, and the definition of the functions required by any LEDA type.

```
⟨bigfloat.h 5⟩ ≡
#ifndef BIGFLOAT_H
#define BIGFLOAT_H
#include <LEDA/integer.h>
#include <math.h>
  enum rounding_modes {
    TO_NEAREST, TO_ZERO, TO_P_INF, TO_N_INF, TO_INF, EXACT };
  enum output_modes { BIN_OUT, DEC_OUT };
  enum special_values { NOT, PZERO, NZERO, PINF, NINF, NAN };
  class bigfloat {
private:
  static long global_prec;
  static rounding_modes round_mode;
  static bool dbool;
  static output_modes output_mode;
  ⟨data members of class bigfloat 9⟩
  ⟨private functions 7⟩
public:
  ~bigfloat() { }
  bigfloat();
  bigfloat(double);
  bigfloat(long);
  bigfloat(int);
  bigfloat(const integer &);
  bigfloat(const integer &s, const integer &e);
  bigfloat(special_values sp);      /* the default copy constructor and assignment
      operator are used (element-wise copy) */
     /* arithmetical functions and operators */
  friend bigfloat add(const bigfloat &, const bigfloat &, long prec = global_prec,
      rounding_modes mode = round_mode, bool &is_exact = dbool);
  friend bigfloat sub(const bigfloat &, const bigfloat &, long prec = global_prec,
      rounding_modes mode = round_mode, bool &is_exact = dbool);
  friend bigfloat mul(const bigfloat &, const bigfloat &, long prec = global_prec,
      rounding_modes mode = round_mode, bool &is_exact = dbool);
  friend bigfloat div(const bigfloat &, const bigfloat &, long prec = global_prec,
      rounding_modes mode = round_mode, bool &is_exact = dbool);
  friend bigfloat sqrt(const bigfloat &, long prec = global_prec, rounding_modes
      mode = round_mode, bool &is_exact = dbool);
  friend bigfloat operator+(const bigfloat &a, const bigfloat &b) {
    return add(a, b); }
```

friend bigfloat operator$-$(**const bigfloat** $\&a$, **const bigfloat** $\&b$) {
  return $sub(a, b)$; }
friend bigfloat operator $*$ (**const bigfloat** $\&a$, **const bigfloat** $\&b$) {
  return $mul(a, b)$; }
friend bigfloat operator/(**const bigfloat** $\&a$, **const bigfloat** $\&b$) {
  return $div(a, b)$; }
friend bigfloat operator$-$(**const bigfloat** $\&$);   /$*$ comparison operators $*$/
friend bool operator$\equiv$(**const bigfloat** $\&$, **const bigfloat** $\&$);
friend bool operator $>$ (**const bigfloat** $\&$, **const bigfloat** $\&$);
friend bool operator$\neq$(**const bigfloat** $\&a$, **const bigfloat** $\&b$)
{ **return** $\neg(a \equiv b)$; }
friend bool operator$\geq$(**const bigfloat** $\&a$, **const bigfloat** $\&b$)
{ **return** $((a > b) \vee (a \equiv b))$; }
friend bool operator $<$ (**const bigfloat** $\&a$, **const bigfloat** $\&b$)
{ **return** $(\neg(a \geq b))$; }
friend bool operator$\leq$(**const bigfloat** $\&a$, **const bigfloat** $\&b$)
{ **return** $(\neg(a > b))$; }   /$*$ rounding $*$/
friend bigfloat $round$(**bigfloat** x, **long** $digits = 0$, **rounding_modes**
  $mode = round\_mode$, **bool** $\&is\_exact = dbool$, **long** $bias = 0$);
  /$*$ tests for special values $*$/
inline friend bool $isNaN$(**const bigfloat** $\&x$)
{ **return** $(x.special \equiv NAN)$; }
inline friend bool $isnInf$(**const bigfloat** $\&x$)
{ **return** $(x.special \equiv NINF)$; }
inline friend bool $ispInf$(**const bigfloat** $\&x$)
{ **return** $(x.special \equiv PINF)$; }
inline friend bool $isnZero$(**const bigfloat** $\&x$)
{ **return** $(x.special \equiv NZERO)$; }
inline friend bool $ispZero$(**const bigfloat** $\&x$)
{ **return** $(x.special \equiv PZERO)$; }
inline friend bool $isZero$(**const bigfloat** $\&x$)
{ **return** $((x.special \equiv PZERO) \vee (x.special \equiv NZERO))$; }
inline friend bool $isInf$(**const bigfloat** $\&x$)
{ **return** $((x.special \equiv PINF) \vee (x.special \equiv NINF))$; }
inline friend bool $isSpecial$(**const bigfloat** $\&x$)
{ **return** $(x.special \neq NOT)$; }   /$*$ mathematical functions $*$/
friend long $sign$(**const bigfloat** $\&x$);
friend long $sign\_of\_special\_value$(**const bigfloat** $\&x$);
inline friend bigfloat abs(**const bigfloat** $\&x$)
{ **return** $(x <$ **bigfloat**$(PZERO)$? $-x : x)$; }
inline friend bigfloat $pow2$(**const integer** $\&p$)
{ **return bigfloat**$(1, p)$; }
inline friend integer $log2$(**const bigfloat** $\&x$)
{ **return** $x.get\_precision(\,) + x.get\_exponent(\,)$; }

```
inline friend integer ceil(const bigfloat &x)
{ return tointeger(x, TO_P_INF); }
inline friend integer floor(const bigfloat &x)
{ return tointeger(x, TO_N_INF); }      /* conversion functions */
double friend todouble(const bigfloat &x);
integer friend tointeger(bigfloat x, rounding_modes rmode = TO_ZERO);
    /* input/output operations */
friend ostream &operator<<(ostream &os, const bigfloat &x);
friend istream &operator>>(istream &is, bigfloat &x);      /* access functions */
long get_precision(void) const
{ return precision; }
integer get_exponent(void) const
{ return exponent; }
integer get_significant(void) const
{ return significant; }      /* functions to set global constants */
static void set_glob_prec(long p)
{ global_prec = p; }
static void set_round_mode(rounding_modes m = TO_NEAREST)
{ round_mode = m; }
static void set_output_mode(output_modes o_mode = DEC_OUT)
{ output_mode = o_mode; }
} ;
⟨ LEDA functions 6 ⟩
#endif
```

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

⟨ LEDA functions 6 ⟩ ≡

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

This code is used in chunk 5.

7. We need to define some additional functions for internal use: *normalize* removes trailing zeroes in the significant, *error_in_rounding* returns the error made in rounding.

⟨ private functions 7 ⟩ ≡

```
void normalize();
bigfloat error_in_rounding(long digits = 0, rounding_modes mode = round_mode,
    bool &is_exact = dbool, long bias = 0);
```

This code is used in chunk 5.

8. The static elements have to be initialized.

$\langle$ Initialization of static members 8 $\rangle \equiv$

long bigfloat :: *global_prec* = 53;
rounding_modes bigfloat :: *round_mode* = *TO_NEAREST*;
bool bigfloat :: *dbool* = *true*;
output_modes bigfloat :: *output_mode* = *DEC_OUT*;

This code is used in chunk 10.

9. The following section contains the data members of data type bigfloat.

$\langle$ data members of class bigfloat 9 $\rangle \equiv$

integer *significant*, *exponent*;
special_values *special*;
long *precision*;

This code is used in chunk 5.

10. File bigfloat.c

The file bigfloat.c has a simple structure. At first the static members of class bigfloat are initialized. Afterwards we define some global identifiers needed by some later defined functions immediately followed by some auxiliary functions, e.g. the *sign* function for the data types integer and long. Then the definition of functions for internal use like *normalize* and the definition of public member functions are given.The member functions come in five big chunks: constructors, general functions (*round*, *conversion*, *sign*, ...), arithmetical functions, comparison operators and input/output operators.

```
⟨ bigfloat.c 10 ⟩ ≡
#include "bigfloat.h"
#include <iostream.h>
#include <strstream.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
  ⟨ Initialization of static members 8 ⟩
  ⟨ global identifiers 58 ⟩
  ⟨ auxiliary functions 16 ⟩
  ⟨ functions for internal use 4 ⟩     /* member functions */
  ⟨ constructors 12 ⟩
  ⟨ general functions 14 ⟩
  ⟨ arithmetical functions 30 ⟩
  ⟨ comparison operators 48 ⟩
  ⟨ input/output operators 82 ⟩
```

11. General Functions

12. Simple Constructors.

A bigfloat can be constructed from an int, a long, a special value, a pair of integers and a double. All but the last one are trivial and given now. The bodies of the constructors from data types int, long and integer are all the same and hence are collected in a special refinement. The default constructor constructs a bigfloat with value *PZERO*.

⟨constructors 12⟩ ≡

```
bigfloat::bigfloat()
{
  special = PZERO;
  exponent = significant = precision = 0;
}
bigfloat::bigfloat(const integer &s, const integer &e)
{
  if (s ≠ 0) {
    significant = s;
    exponent = e;
    special = NOT;
    precision = s.length();
  }
  else {
    special = PZERO;
    exponent = significant = precision = 0;
  }
}
bigfloat::bigfloat(special_values sp) { special = sp; }
bigfloat::bigfloat(const integer &a)
{ ⟨constructor body for integer data type 13⟩ }
bigfloat::bigfloat(long a)
{ ⟨constructor body for integer data type 13⟩ }
bigfloat::bigfloat(int a)
{ ⟨constructor body for integer data type 13⟩ }
```

See also chunk 59.

This code is used in chunk 10.

13.⟨constructor body for integer data type 13⟩ ≡

```
special = NOT;
significant = a;
exponent = 0;
precision = significant.length();
if (significant ≡ 0) special = PZERO;
```

This code is used in chunk 12.

14. The function *sign* computes the sign of an instance of data type bigfloat. It returns a -1 if the bigfloat represents an negative value, a zero if it is zero and otherwise a 1.

⟨general functions 14⟩ ≡

```
long sign(const bigfloat &x)
{
  switch (x.special) {
  case NOT: return ::sign(x.significant);
  case PZERO: case NZERO: return 0;
  case PINF: return 1;
  case NINF: return −1;
  case NAN: error_handler(1, "sign:␣NaN␣has␣no␣sign");
  }
}
```

See also chunks 15, 18, 28, and 67.

This code is used in chunk 10.

15. The function *sign_of_special_value* returns a nonzero long value. The function enables the user to determine the sign of 0 or ∞. If this function is called with argument *NAN* an error message will be thrown out. The call of function *normalize* at the beginning guarantees that a bigfloat with value 0 is represented by a special value. On non special values this function performs the same actions as the *sign* function for data type bigfloat.

⟨general functions 14⟩ +≡

```
long sign_of_special_value(const bigfloat &x)
{
  ((bigfloat &) x).normalize();
  if (x.special ≡ NAN) error_handler(1,
        "sign_of_special_value:␣want␣a␣special␣value␣but␣not␣NaN");
  if (x.special ≡ NOT) return sign(x);
  if (x.special ≡ PZERO ∨ x.special ≡ PINF) return 1;
  else return −1;
}
```

16. Before we come to the core of the bigfloat implementation, we list some useful functions to compute the maximum of two longs, the square of two integers, the binary logarithm of a double and the sign for types long and int, respectively.

⟨auxiliary functions 16⟩ ≡

```
inline long max(long l1, long l2) { return (l1 > l2 ? l1 : l2); }
inline integer sq(const integer &op) { return op ∗ op; }
inline double log2(double d) { return log(d)/log(2); }
inline long sign(int i) { if (i ≡ 0) return 0; else return (i > 0 ? 1 : −1); }
inline long sign(long i) { if (i ≡ 0) return 0; else return (i > 0 ? 1 : −1); }
```

See also chunk 86.

This code is used in chunk 10.

17. Rounding

The *round* function is the central tool for the implementation of bigfloat arithmetic. Recall that $round(x, digits, mode, is\_exact, bias)$ rounds the number

$$(significant + bias \cdot \epsilon) \cdot 2^{exponent}$$

to *digits* binary digits in rounding mode *mode*, where ϵ is a positive infinitesimal and bias is out of $\{-1, 0, 1\}$. *is_exact* is set to *true* if the rounded number equals to x.

18. Now we outline the implementation of function *round*. At first the bigfloat is normalized in order to avoid final zeros of the significant. Then we distinguish cases. It might be that the *significant* has more places than specified by parameter *digits*. In this case the *bias* can be neglected, with the one exception of the *TO_NEAREST* mode (details see below). In all cases except for *EXACT* the *significant* is rounded to *digits* places and the exponent is adapted accordingly. On the other hand, it might be that the length of the *significant* is less than *digits*. In that case the return value depends on the bias (again with the exception of mode *TO_NEAREST*).

```
⟨general functions 14⟩ +≡
  bigfloat round(bigfloat x, long digits, rounding_modes mode, bool &is_exact, long
          bias)
  {
    x.normalize();
    if (isSpecial(x) ∨ (mode ≡ EXACT)) {
      is_exact = true;
      return x;
    }
    int test = 0;
    long shift;
    if (digits < x.precision) {
      switch (mode) {
      case TO_NEAREST:
        ⟨round to nearest 20⟩
        break;
      case TO_ZERO:
        ⟨round to zero 22⟩
        break;
      case TO_P_INF:
        ⟨round to plus infinity 24⟩
        break;
      case TO_N_INF:
        ⟨round to minus infinity 25⟩
        break;
      case TO_INF:
        ⟨round to infinity 23⟩
        break;
      }
```

```
    x.exponent += (x.precision − digits);
  }
  else if (bias ≠ 0) {⟨ bias rounding 26 ⟩}
  is_exact = (digits ≥ x.precision) ∧ (bias ≡ 0);
  x.normalize( );
  return x;
}
```

19. We need a function that cuts an integer to an arbitrary amount of digits.

```
⟨ functions for internal use 4 ⟩ +≡
  void cut(integer &b, long prec) { b = (b ≫ (b.length( ) − prec)); }
```

20. First we implement rounding in the case that the *significant* of the normalized bigfloat has more than *digits* places and hence the rounded value is not equal to the exact value.

We start with the rounding case *TO_NEAREST*. Write the *significant* as $x_1 \cdot 2^{precision-digits} + x_2$ and recall that x_2 is odd since x is normalized. If x_2 starts with a zero then x_1 is the rounded *significant*.If x_2 starts with a one and has more than one digit, then $x_1 + sign(x_1)$ is the rounded significant. If x_1 has only one digit of value one the *bias* decides the rounding.

```
⟨ round to nearest 20 ⟩ ≡
  cut(x.significant, digits + 1);
  test = ⟨ significant is even 21 ⟩;
  cut(x.significant, digits);
  if (¬test) {      /* x2 starts with a one */
    if ((x.precision > digits + 1) ∨ (::sign(bias) ≡ ::sign(x.significant)) ∨ ((bias ≡
          0) ∧ ¬⟨ significant is even 21 ⟩)) {
        /* x1 + sign(x1) is the rounded value */
      if (::sign(x.significant) > 0) x.significant++; else x.significant−−;
    }
  }
```

This code is used in chunk 18.

21. It is simple to test whether the *significant* of x is even.

```
⟨ significant is even 21 ⟩ ≡
  ((x.significant & integer(1)) ≡ 0)
```

This code is used in chunk 20.

22. In the *TO_ZERO* case we cut the significant to the wanted amount of digits.

```
⟨ round to zero 22 ⟩ ≡
  cut(x.significant, digits);
```

This code is used in chunk 18.

23. In the *TO_INF* case we always add ± 1 because we already know that the rounded value is not exact.

$\langle$ round to infinity 23 $\rangle \equiv$

```
cut(x.significant, digits);
if (::sign(x.significant) > 0) x.significant++; else x.significant--;
```

This code is used in chunk 18.

24. The next case is *TO_P_INF*. Here the *significant* is rounded up. If the number is negative this is done by cutting *significant* to length *digits*. Otherwise we increment the *significant* of x after the cutting.

$\langle$ round to plus infinity 24 $\rangle \equiv$

```
cut(x.significant, digits);
if (::sign(x.significant) > 0) x.significant++;
```

This code is used in chunk 18.

25. The *TO_N_INF*-mode is the opposite case to *TO_P_INF*.

$\langle$ round to minus infinity 25 $\rangle \equiv$

```
cut(x.significant, digits);
if (::sign(x.significant) < 0) x.significant--;
```

This code is used in chunk 18.

26. Now we come to the cases where the wanted *precision* is less or equal to the original *precision* and *bias* is crucial. Again we consider the rounding modes separately. For mode *TO_NEAREST* there is nothing to do. In the other cases, it depends on the sign of the bias whether we have to do nothing, or else we have to add a 1 or -1 at the $digits^{th}$ place.

$\langle$ bias rounding 26 $\rangle \equiv$

```
shift = digits - x.precision;
long bf_sign = ::sign(x.significant);
switch (mode) {
case TO_ZERO:
  if (::sign(bias) ≠ bf_sign) {
    ⟨ shift significant 27 ⟩
    if (bf_sign ≡ 1) x.significant--; else x.significant++;
  }
  break;
case TO_INF:
  if (::sign(bias) ≡ bf_sign) {
    ⟨ shift significant 27 ⟩
    if (bf_sign ≡ 1) x.significant++; else x.significant--;
  }
  break;
case TO_P_INF:
  if (::sign(bias) ≡ 1) {
    ⟨ shift significant 27 ⟩
```

```
      x.significant++;
    }
    break;
  case TO_N_INF:
    if (::sign(bias) ≡ −1) {
      ⟨shift significant 27⟩
      x.significant−−;
    }
    break;        /* in the cases TO_NEAREST and EXACT we have to do nothing */
  case TO_NEAREST: case EXACT: ;
  }
```

This code is used in chunk 18.

27. When shifting the *significant* left by *shift* digits we have to lower the bigfloat's exponent by the same amount.

```
⟨shift significant 27⟩ ≡
  if (shift > 0) {
    x.significant = x.significant ≪ shift;
    x.exponent −= shift;
  }
```

This code is used in chunk 26.

28. A bigfloat x can also be rounded to integer format by the function *to_integer*. If x represents a nonzero special value, we throw out an error message. Now let us assume that x is non special. We round x to $length = max(x.precision + x.exponent, 1)$ places. The choice of *length* guarantees that the rounding returns an integer value nearest to the original fractional value of x. In particular, the *exponent* of x is nonnegative after rounding. Due to the choice of *length*, x is representable with an *exponent* of value zero. If the final *normalize* in function *round* produced an *exponent* ≥ 0 the *significant* of x has to be shifted to the left by this amount of binary places. Afterwards *x.significant* might be returned as the result.

```
⟨general functions 14⟩ +≡
  integer tointeger(bigfloat x, rounding_modes rmode)
  {
    if ((isNaN(x)) ∨ (isInf(x))) error_handler(1,
          "tointeger␣:␣special␣values␣cannot␣be␣converted␣to␣integer");
    if (¬(x.exponent + (integer) x.significant.length()).islong())
      error_handler(1, "tointeger␣:␣(exp+sig␣len)␣has␣to␣be␣in␣long␣range");
    long length = max(1, x.precision + x.exponent.tolong());
    x = round(x, length, rmode);
    if (isZero(x)) return integer(0);
    else return (x.significant ≪ x.exponent.tolong());
  }
```

29. Arithmetical functions

30. The Add-Function.

In the following sections we explain the arithmetical functions *add*, *sub*, *mul*, *div*, and *sqrt*. We start right off with the *add* function.

Let a and b be the operands of the addition. At first we make sure that the binary logarithm of a (rounded up to the next integer) is not less than the binary logarithm of b. Then we compute bigfloats *sum* and *error* such that we have the exact equality

$$a + b = sum + error.$$

Here *error* is bounded by half the least significant digit of *sum* and $error \equiv 0$ for $mode \equiv EXACT$. Finally we give back the rounded value of *sum*. Procedure *round* only needs to know the sign of *error*.

```
⟨ arithmetical functions 30 ⟩ ≡
  bigfloat add(const bigfloat &x, const bigfloat &y, long prec, rounding_modes
          mode, bool &is_exact)
  {
    bigfloat a, b;
    ⟨ handle special cases 35 ⟩
    ⟨ find bigger operand 31 ⟩      /* now ⌈log_2 a⌉ ≥ ⌈log_2 b⌉ */
    bigfloat sum, error;
    ⟨ compute sum and error 32 ⟩
    return round(sum, prec, mode, is_exact, sign(error));
  }
```

See also chunks 36, 37, 39, 43, and 47.

This code is used in chunk 10.

31. It is helpful to know the *bigger* operand. To decide which operand is bigger we compute for x and y the sums of exponent and precision, called *log_x* and *log_y*. The difference *diff* of these quantities must not be confused with the difference of the exponents, *exp_diff*.

```
⟨ find bigger operand 31 ⟩ ≡
  bigfloat *a_ptr, *b_ptr;
  integer log_x = x.exponent + (integer) x.precision;
  integer log_y = y.exponent + (integer) y.precision;
  integer diff = log_x − log_y;
  if (diff ≥ 0) { a_ptr = (bigfloat *) &x; b_ptr = (bigfloat *) &y; }
  else { a_ptr = (bigfloat *) &y; b_ptr = (bigfloat *) &x; }
  a = *a_ptr;
  b = *b_ptr;
  diff.absolute();
  integer exp_diff = a.exponent − b.exponent;
```

This code is used in chunk 30.

32. Often it is unnecessary to compute the exact sum. If b has no influence on the addition's result it suffices to set *sum* to a and *error* to b. This is the case if b is less than the least significant digit of a and also less than the $(prec+1)^{th}$ digit of a. (Here we can assume that a has at least $prec+1$ digits by conceptually shifting left, if it has less digits.) These restrictions can be formulated as $diff > a.precision$ and $diff > (prec+1)$. Furthermore the rounding mode must be different from $EXACT$. If one of the conditions is violated we compute the addition exactly.

⟨ compute sum and error 32 ⟩ ≡
```
if ((mode ≠ EXACT) ∧ (diff > max(prec + 1, a.precision)))
{
   sum = a;
   error = b;
}
else {
   ⟨ exact addition 34 ⟩
}
```
This code is used in chunk 30.

33. The exact addition of two bigfloat numbers can easily be attributed to integer addition. The precondition for the use of integer arithmetic is that the two bigfloat operands have equal exponents. In this case the result can be calculated as follows:

⟨ calculate sum 33 ⟩ ≡
```
sum = bigfloat(a.significant + b.significant, a.exponent);
```
This code is used in chunk 34.

34. If $a.exponent > b.exponent$ we shift $a.significant$ leftwards by $exp\_diff \equiv a.exponent - b.exponent$ binary places. Similarly, if $b.exponent > a.exponent$ we shift $b.significant$ leftwards by $-exp\_diff$ places. In both cases the exponents have to be set to the smallest exponent of a and b.

⟨ exact addition 34 ⟩ ≡
```
if (¬exp_diff.islong())
   error_handler(1, "bigfloat::add()␣:␣exponential␣differenc\
      e␣has␣to␣be␣in␣long␣range");
if (exp_diff > 0) {
   a.significant = a.significant ≪ exp_diff.tolong();
   a.exponent = b.exponent;
}
else {
   b.significant = b.significant ≪ (−exp_diff).tolong();
   b.exponent = a.exponent;
}
⟨ calculate sum 33 ⟩
```
This code is used in chunk 32.

35. The rules to handle special cases in addition are simple. The result is NaN if one of the operands is NaN or for the sums $\infty + (-\infty)$ and $(-\infty) + \infty$. In the other cases we return the sum of x and y.

⟨ handle special cases 35 ⟩ ≡
```
((bigfloat &) x).normalize();
((bigfloat &) y).normalize();
if (isSpecial(x) ∨ isSpecial(y)) {
  if (isNaN(x) ∨ isNaN(y)) return bigfloat(NAN);
  if (isZero(x)) return y;
  if (isZero(y)) return x;
  if (isInf(x) ∧ isInf(y)) {
    if (sign_of_special_value(x) ≡ sign_of_special_value(y)) return x;
    else return bigfloat(NAN);
  }
  if (isInf(x)) return x;      /* it is obvious that y has to be ∞ */
  return y;
}
```
This code is used in chunk 30.

36. The Sub-Function.

This function is quite easy because it can be reduced to the *add* function.

⟨ arithmetical functions 30 ⟩ +≡
```
bigfloat sub(const bigfloat &a, const bigfloat &b, long prec, rounding_modes
    mode, bool &is_exact)
{
  return add(a, −b, prec, mode, is_exact);
}
```

37. The Mul-Function.

The multiplication is always done by first computing the exact result and rounding afterwards, in contrast to the procedure for addition. Computing the exact result is done by simply multiplying the significants of the operands and adding their exponents.

⟨ arithmetical functions 30 ⟩ +≡
```
bigfloat mul(const bigfloat &a, const bigfloat &b, long prec, rounding_modes
    mode, bool &is_exact)
{
  ⟨ special cases for mul 38 ⟩
  bigfloat result(a.significant ∗ b.significant, a.exponent + b.exponent);
  return round(result, prec, mode, is_exact);
}
```

38. We come to the special case treatment of *mul*. The result is *NaN* if one of the operands is *NaN* or else, if one of the operands is zero and the other one infinity. Otherwise, if one operand is zero, we return zero, and if one of the operands is infinity, we return infinity. Here the sign of the return value is the product of the operand signs.

⟨ special cases for mul 38 ⟩ ≡

```
long sign_result;
((bigfloat &) a).normalize( );
((bigfloat &) b).normalize( );
if ((isSpecial(a)) ∨ (isSpecial(b))) {
  if ((isNaN(a)) ∨ (isNaN(b))) return bigfloat(NAN);
  if ((isZero(a) ∧ isInf(b)) ∨ (isInf(a) ∧ isZero(b))) return bigfloat(NAN);
  sign_result = sign_of_special_value(a) * sign_of_special_value(b);
  if (isZero(a) ∨ isZero(b)) {
    if (sign_result ≡ 1) return bigfloat(PZERO);
    else return bigfloat(NZERO);
  }
  if (isInf(a) ∨ isInf(b)) {
    if (sign_result ≡ 1) return bigfloat(PINF);
    else return bigfloat(NINF);
  }
}
```

This code is used in chunk 37.

39. The Div-Function.

One important difference between the division and other arithmetical functions is that the exact calculation of a division in bigfloat format is impossible. Instead we use inexact integer division of the *significant*s to approximate the result up to $(prec+1)$ digits [1]. Here it may be necessary to shift the *significant* of the dividend to the left. We also compute the sign of the division remainder in the variable *bias*. Then the correctly rounded division result can be obtained by calling the *round* function with this bias.

⟨ arithmetical functions 30 ⟩ +≡

```
bigfloat div(const bigfloat &a, const bigfloat &b, long prec, rounding_modes
        mode, bool &is_exact)
{
  bigfloat result;
  long bias;
  ⟨ special cases for div 42 ⟩
  ⟨ shift dividend's significant 40 ⟩
  ⟨ compute approximative result 41 ⟩
  return round(result, prec, mode, is_exact, bias);
}
```

40. We first calculate if and by how many digits the dividend a has to be shifted. Let S_a and S_b be the *significant*s of the operands with binary lengths l_a and l_b. We suppose that S_a and S_b are nonzero. Then we have $2^{l_a-1} \leq S_a < 2^{l_a}$ and $2^{l_b-1} \leq S_b < 2^{l_b}$ which implies

$$2^{l_a-l_b-1} < S_a/S_b < 2^{l_a-l_b+1}.$$

[1] we need one extra digit to guarantee exact rounding

Hence the precision of S_a/S_b is at least $l_a - l_b$. Therefor we need $l_a - l_b \geq prec + 1$ and we have to shift the *significant* of a by d digits, if $d = l_b - l_a + prec + 1 > 0$.

$\langle$ shift dividend's significant 40 $\rangle \equiv$

```
bigfloat aa = a;
long d = prec + b.significant.length() − a.significant.length() + 1;
if (d > 0) {
   aa.significant = aa.significant ≪ d;
   aa.exponent −= d;
}
```

This code is used in chunk 39.

41. Computing the approximation of the result is now easy. We simply do a Euclidean division of S_a and S_b, that is, $S_a = S_b \cdot S_r + R$ where R is the division remainder. This is equivalent to

$$S_a/S_b = S_r + R/S_b$$

and hence the bias is given by the sign of R/S_b.

$\langle$ compute approximative result 41 $\rangle \equiv$

```
result.special = NOT;
result.significant = aa.significant/b.significant;
result.exponent = aa.exponent − b.exponent;
result.precision = result.significant.length();
integer R = aa.significant − b.significant ∗ result.significant;
if (R ≠ 0) is_exact = false;
if (mode ≡ EXACT) mode = TO_NEAREST;
bias = sign(R) ∗ sign(b.significant);
```

This code is used in chunk 39.

42. The special case handling for division is very similar to that for multiplication. We omit the details.

$\langle$ special cases for div 42 $\rangle \equiv$

```
((bigfloat &) a).normalize();
((bigfloat &) b).normalize();
if ((isSpecial(a)) ∨ (isSpecial(b))) {
   long sign_result;
   if ((isNaN(a)) ∨ (isNaN(b))) return bigfloat(NAN);
   if (((isZero(a)) ∧ (isZero(b))) ∨ ((isInf(a)) ∧ (isInf(b)))) return bigfloat(NAN);
   sign_result = sign_of_special_value(a) ∗ sign_of_special_value(b);
   if ((isInf(a)) ∨ (isZero(b))) {
      if (sign_result ≡ 1) return bigfloat(PINF);
      else return bigfloat(NINF);
   }    /∗ it is clear that isZero(a) ∨ ( isInf(b) ∗/
   if (sign_result ≡ 1) return bigfloat(PZERO);
   else return bigfloat(NZERO);
}
```

This code is used in chunk 39.

43. The Sqrt-Function.

We reduce the square root operation for the data type bigfloat to the one of integers. The function is splitted into three main parts:

1. treatment of special cases
2. calculation of the square root
3. rounding

⟨ arithmetical functions 30 ⟩ +≡

bigfloat *sqrt*(**const bigfloat** &*a*, **long** *prec*, **rounding_modes** *mode*, **bool** &*is_exact*)
{
  bigfloat *result*;
  integer *s*;
  ⟨ special cases of sqrt 46 ⟩
  ⟨ calculate sqrt 44 ⟩
  ⟨ rounding of sqrt 45 ⟩
}

44. We rely on the fact that the LEDA data type integer offers a function $n.sqrt() = \lfloor\sqrt{n}\rfloor$. Furthermore, we know that the delivered bigfloat is positive and non special since we successfully passes the special case section. We now have to ensure that resulting bigfloat has precision of *prec* digits.
Let now l be the *significant*'s length, $sig = (1+\delta)\cdot 2^{l-1}$ be the *significant* of a with $0 \le \delta < 1$, exp its exponent and let k be the smallest value with such that

1. $exp - k$ is even,
2. $2^{prec-1} \le \sqrt{sig\cdot 2^k} < 2^{prec}$

Let $s = \lfloor\sqrt{sig\cdot 2^k}\rfloor$. Then $\sqrt{a} = \sqrt{sig\cdot 2^k\cdot 2^{exp-k}} = (s+\epsilon)\cdot 2^{(exp-k)/2}$ for some ϵ with $0 \le \epsilon < 1$. With inequality (2) it follows that $2\cdot(prec-1) \le \log_2 sig\cdot 2^k < 2\cdot prec$. Since $sig = (1+\delta)\cdot 2^(l-1)$ it follows that $\log_2 sig = (l-1)+\log_2(1+\delta)$. Thus the following applies:

$$2\cdot prec-2 \;\le\; (k+l-1)+\log_2(1+\delta) \;<\; 2\cdot prec$$

Notice that $\log_2(1+\delta) < 1)$. Hence we can simplify the expression:

$$2\cdot prec-2 \le \; k+l-\epsilon \;<\; 2\cdot prec$$

where $0 \le \epsilon < 1$.Since k, l and $prec$ are integer values we conclude that

$$\begin{array}{rcccl} 2\cdot prec-2 & < & k+l & < & 2\cdot prec+1 \\ 2\cdot prec-l-2 & < & k & < & 2\cdot prec-l+1 \\ 2\cdot prec-l-1 & \le & k & \le & 2\cdot prec-l \end{array}$$

Hence we choose $k = 2\cdot prec - l$. If $exp - k$ is odd we decrement k by one.

⟨calculate sqrt 44⟩ ≡

```
long k = max(0, 2 * prec - a.significant.length());     /* check if exp - k is odd */
if (((a.exponent - k) % 2) ≠ 0) k--;
integer r = a.significant << k;
s = sqrt(r);
```

This code is used in chunk 43.

45. We come to the rounding of the result. If $s^2 = sig \cdot 2^k$ the result is $s \cdot 2^{(exp-k)/2}$. Hence we assume that $s < \sqrt{sig \cdot 2^k}$. We differ between 3 rounding cases:

1. <u>*TO_ZERO*, *TO_N_INF*</u>: the result is $s \cdot 2^{(exp-k)/2}$
2. <u>*TO_INF*, *TO_P_INF*</u>: the result is $(s+1) \cdot 2^{(exp-k)/2}$
3. <u>*TO_NEAREST*, *EXACT*</u>: We have to decide whether s or $s+1$ is the nearest value to $\sqrt{sig \cdot 2^k}$. If s is the best approximation we have $\sqrt{sig \cdot 2^k} - s < (s+1) - \sqrt{sig \cdot 2^k}$ which is equivalent to $4 \cdot sig \cdot 2^k < 4 \cdot s^2 + 4 \cdot s + 1$.

Remember that $r = sig \cdot 2^k$.

⟨rounding of sqrt 45⟩ ≡

```
integer s2 = s * s;
if ((s2 ≡ r) ∨ (mode ≡ TO_ZERO) ∨ (mode ≡ TO_N_INF))
  return bigfloat(s, (a.exponent - k)/2);
if ((mode ≡ TO_INF) ∨ (mode ≡ TO_P_INF))
  return bigfloat(s + 1, (a.exponent - k)/2);
    /* mode is either TO_ZERO or EXACT */
if (4 * r < 4 * s2 + 4 * s + 1) return bigfloat(s, (a.exponent - k)/2);
else return bigfloat(s + 1, (a.exponent - k)/2);
```

This code is used in chunk 43.

46. The rules for the special case treatment are simple. If it is strictly negative we return *NaN*. Otherwise, if it is any special value beside $-\infty$, we return the same value.

⟨special cases of sqrt 46⟩ ≡

```
((bigfloat &) a).normalize();
if (sign(a) < 0) { is_exact = false; return bigfloat(NAN); }
if (isSpecial(a)) {
  if (isZero(a)) is_exact = true; else is_exact = false;
  return a;
}
```

This code is used in chunk 43.

47. We now come to the implementation of the unary minus operator.

⟨ arithmetical functions 30 ⟩ +≡

```
bigfloat operator-(const bigfloat &a)
{
  if (isSpecial(a)) {
    if (isZero(a)) return bigfloat(PZERO);
    if (isInf(a)) return bigfloat(PINF);
    return bigfloat(NAN);
  }
  return bigfloat(-a.significant, a.exponent);
}
```

48. Comparison Operators

We still have to implement the operators $\equiv$ and $>$. Remember that the other comparison operators have been reduced to these two cases. Let us begin with operator $\equiv$. We first normalize [2] the operands to get a unique representation and afterwards check for special cases. Then we only have to compare significant and exponent.

⟨ comparison operators 48 ⟩ ≡

```
bool operator≡(const bigfloat &a, const bigfloat &b)
{
  ((bigfloat &) a).normalize( );
  ((bigfloat &) b).normalize( );
  ⟨ special case checking for operator ≡ 49 ⟩
  return ((a.significant ≡ b.significant) ∧ (a.exponent ≡ b.exponent));
}
```

See also chunk 50.

This code is used in chunk 10.

49. The rules of special-case checking can be summarized as follows. Comparisons with a *NaN* not allowed. If both operands are zero, *true* is returned. Otherwise, *true* is returned if and only if the operands represent the same special value.

⟨ special case checking for operator ≡ 49 ⟩ ≡

```
if (isSpecial(a) ∨ isSpecial(b)) {
  if (isZero(a) ∧ isZero(b)) return true;
  if (isNaN(a) ∨ isNaN(b))
    error_handler(1, "bigfloat::operator␣==␣:␣NaN␣case␣occurred");
  return (a.special ≡ b.special);
}
```

This code is used in chunk 48.

50. We come to **operator** $>$. For non-special bigfloats we first check the signs of the operands. If the operand's signs are not equal the result may be easily computed. In case of equal signs we compare the binary lengths of the arguments. Let now $a = s_a \cdot 2^{e_a}$ and $b = s_b \cdot 2^{e_b}$. Furthermore, let l_a and l_b be the binary length of s_a and s_b respectively. We know that $2^{l_a-1} \le s_a < 2^{l_a}$ and $2^{l_b-1} \le s_b < 2^{l_b}$. Let $bl_a = l_a + e_a$ and $bl_b = l_b + e_b$. It follows that $a > b$ if $2^{l_a-1} \cdot 2^{e_a} > 2^{l_b} \cdot 2^{e_b}$. That is $a > b$ if $bl_a - bl_b > 1$. Analogously, one can see that $a \le b$ if $bl_a - bl_b \le -1$. So we have to watch $bl_a - bl_b$. If none of the latter cases apply we find out the operand with the bigger exponent and shift its significant by the exponential difference to the left. Afterwards we return the integer comparison of the significants.

⟨ comparison operators 48 ⟩ +≡

```
bool operator > (const bigfloat &a, const bigfloat &b)
{
```

[2] Since *normalize* is a not a constant member function we first cast the operands to type **bigfloat** &. This is allowed since *normalize* does not change the value of a bigfloat but only its representation. Here we use the idea of logical constance.

((**bigfloat** &) a).*normalize*( );
((**bigfloat** &) b).*normalize*( );
⟨ special case checking of operator $>$ 51 ⟩
int *sign_a* $=$ ::*sign*(*a.significant*);
int *sign_b* $=$ ::*sign*(*b.significant*);
if (*sign_a* $\neq$ *sign_b*) **return** (*sign_a* $>$ *sign_b*);
integer *bl_diff* $= (a.exponent + a.precision) - (b.exponent + b.precision)$;
if (*bl_diff* > 1) **return** 1;
else if (*bl_diff* ≤ -1) **return** 0;
integer *sig*, *diff* $= a.exponent - b.exponent$;
if (::*sign*(*diff*) ≥ 0) {
  sig $= a.significant \ll (diff.tolong())$;
  return (*sig* $>$ *b.significant*);
}
else {
  sig $= b.significant \ll ((-diff).tolong())$;
  return (*a.significant* $>$ *sig*);
}
}

51. Finally we come to the special case checking for operator $>$. As before, *NaN* comparisons are not allowed. If the $\equiv$ operator returns true, operator $>$ returns false. The remaining cases are straightforward.

⟨ special case checking of operator $>$ 51 ⟩ $\equiv$
if (*isSpecial*(a) $\vee$ *isSpecial*(b)) {
  if (*isNaN*(a) $\vee$ *isNaN*(b))
    error_handler(1, `"bigfloat::operator␣>␣:␣NaN␣case␣occurred!"`);
  if (*isZero*(a) $\wedge$ *isZero*(b)) **return** *false*;
  if (*ispInf*(a)) **return** $\neg$*ispInf*(b);
  if (*isnInf*(b)) **return** $\neg$*isnInf*(a);
  return (*sign*(a) $>$ *sign*(b));
}
This code is used in chunk 50.

52. Conversion between the data types bigfloat and double

It is possible to convert a double into a bigfloat and vice versa. If the significant of the bigfloat has more than 53 bits or if its exponent is too large there will be a loss of information. For a double d we will always guarantee the consistency property **bigfloat**$(d).todouble(\,) \equiv d$.

53. Some facts about doubles.

A double is specified by a sign $s \in \{0,1\}$, an exponent $e \in [0,2047]$ and a binary fraction $F = f_1 f_2 \ldots f_{52}$ with $f_i \in 0,1$ for all $1 \le i \le 52$. For $0 < e < 2047$ the triple (s,f,e) represents the number

$$(-1)^s \cdot 1.f \cdot 2^{e-1023}.$$

Such a number is called *normalized.* If e is zero the number is called *denormalized.* and the value represented by (s,f,e) is

$$(-1)^s \cdot 0.f \cdot 2^{-1023}.$$

Denormalized numbers are smaller than the smallest normalized number $double\_min = 2^{-1022}$. Due to the limited exponent range doubles can over- and underflow. To get more security in arithmetical operations there are the values $\pm\infty$ signaling overflow and the value NaN signaling invalid operations like $0 \cdot \infty$. $\pm\infty$ are represented by $s \in \{0,1\}$, $e = 2047$ and $f = 0$, and NaN is represented by any triple (s,e,f) with $e = 2047$ and $f \neq 0$.

54. The function *compose_parts* composes the parameters (s,e,f) to a double value. The sign s is delivered in *sign_1*, and *exp_11* is the exponent e. Since we use 32 bit wide longs, the 52 bit of f are split into the lower part *least_sig_32* and the higher part *most_sig_20*. The resulting double is made of two longs, the higher 32 bit containing *sign_1*, *exp_11* and *most_sig_20*, and the lower 32 bit of *least_sig_32*.

⟨functions for internal use 4⟩ +≡

```
double compose_parts(long sign_1, long exp_11, long most_sig_20, long least_sig_32)
{
  double a;
  long high32 = 0;
  ⟨calculate high32 55⟩
  ⟨put it all together 56⟩
  return a;
}
```

55. First we compute the higher 32 bit of the double.

⟨calculate high32 55⟩ ≡

```
if (sign_1) high32 = high32 | #80000000;
exp_11 = exp_11 ≪ 20;
high32 = high32 | exp_11;
high32 = high32 | most_sig_20;
```

This code is used in chunk 54.

56. To compose the two long parts to one double value we use pointer arithmetic. For this we need a pointer to a long variable that initially points to the beginning of the resulting double a. Since we would like to achieve machine independent code we have to care for different byte ordering mechanisms. Sun Sparc stations use *BIG ENDIAN* byte ordering which means that always the higher part is saved before the lower part in memory. Intel PCs use *LITTLE ENDIAN* byte ordering in which the lower part is saved first.

```
⟨put it all together 56⟩ ≡
  long *p;
  p = (long *) &a;
#ifndef LITTLE_ENDIAN
  (*p) = high32; p++; (*p) = least_sig_32;
#else
  (*p) = least_sig_32; p++; (*p) = high32;
#endif
```

This code is used in chunk 54.

57. The mathematical function *pow2* for doubles that computes 2^{exp} for a given exponent *exp* can be realized efficiently using the new function *compose_parts*.

```
⟨functions for internal use 4⟩ +≡
  double pow2(long exp) { return compose_parts(0, exp + 1023, 0, 0); }
```

58. Now we are able to define the special double values that we need in the following sections.

```
⟨global identifiers 58⟩ ≡    /* since we need the pow2 and compose_parts functions we
        have to use prototyping */
  double compose_parts(long, long, long, long);
  double pow2(long);
  const double double_min = pow2(−1022);
  const double NaN_double = compose_parts(0, 2047, 0, 1);
  const double pInf_double = compose_parts(0, 2047, 0, 0);
  const double nInf_double = −pInf_double;
  const double pZero_double = compose_parts(0, 0, 0, 0);
  const double nZero_double = compose_parts(1, 0, 0, 0);
```

See also chunks 66 and 74.

This code is used in chunk 10.

59. The next constructor transforms a double d into a bigfloat. First we check whether d is denormalized. In the case that d is denormalized, we correct the exponent of d to make it normalized (if it is nonzero) but set a flag that allows us to take back our changes later. After that we split d into a high and a low part. From these two parts we compute the sign, significant and exponent of d.

⟨ constructors 12 ⟩ +≡

```
bigfloat::bigfloat(double d)
{
  int sign;
  long flag = 0;
  unsigned long mh = 0, ml = 0;
  long *p;
  ⟨ check for denormalized number 60 ⟩
  ⟨ determine high and low part 61 ⟩
  ⟨ get the double's sign 62 ⟩
  ⟨ get the significant's value 63 ⟩
  ⟨ get the exponent's value 64 ⟩
  ⟨ check for special values 65 ⟩normalize( );
}
```

60. If d is smaller than *double_min* then it is denormalized and we multiply d with 2^{52} to get a normalized number.

⟨ check for denormalized number 60 ⟩ ≡

```
if (fabs(d) < double_min) {
  d = d * pow2(52);
  flag = 1;
}
```

This code is used in chunk 59.

61. We split the 64 bit wide double representation in two 32 bit wide longs. To do this we use a pointer to a long value and assign the casted address of the double value to it. The long pointer works like an **array**. If the *BIG ENDIAN* byte ordering is active, the first component holds the higher 32 bit part *mh* and if the *LITTLE ENDIAN* byte ordering is active, the first component holds the lower 32 bit part *ml*.

⟨ determine high and low part 61 ⟩ ≡

```
  p = (long *) &d;
#ifndef LITTLE_ENDIAN
  mh = *p; p++; ml = *p;
#else
  ml = *p; p++; mh = *p;
#endif
```

This code is used in chunk 59.

62. The *sign* of a double is denoted by its highest bit.

⟨ get the double's sign 62 ⟩ ≡

```
if (mh & #80000000) sign = −1;
else sign = 1;
```

This code is used in chunk 59.

63. The significant of the double is composed out of ml and the least 20 bits of mh. As we have ensured normalized representation in the beginning of this function we have to keep in mind that the significant represents the binary fraction and that there is an implicit one bit that leads this fraction.

$\langle$ get the significant's value 63 $\rangle \equiv$

```
long sig = 0;    /* get the significant bits out of the highword */
sig = mh & #000fffff;
    /* we have ensured that d is normalized ⇒ add the leading one bit */
sig = sig | #00100000;    /* compose significant */
significant = integer(sig);
significant = significant ≪ 32;
significant = significant + integer(ml);
if (sign ≡ −1) significant = −significant;
```

This code is used in chunk 59.

64. The double exponent takes the bits 2 to 12 of mh. We use bit manipulation to extract this value out of mh. To transform the double exponent into the exponent of the resulting bigfloat, we have to correct its value by

1. the bias -1023,
2. $-(significant.length() - 1) \equiv -52$, due to the bigfloat's representation
3. -52, if d was denormalized

$\langle$ get the exponent's value 64 $\rangle \equiv$

```
long e = 0;    /* get the 11 bits of the exponent */
e = mh & #7ff00000;    /* shift the result to get the right value */
e = e ≫ 20;    /* subtract bias and 52 */
e −= 1075;    /* subtract 52 if d was denormalized */
if (flag) e −= 52;
exponent = e;
```

This code is used in chunk 59.

65. Finally, we have to check whether d was in a special state.

$\langle$ check for special values 65 $\rangle \equiv$

```
special = NOT;
if (e ≡ 972)    /* Inf or NaN-Case */
{
  if (significant ≡ 0) special = (sign > 0) ? PINF : NINF;
  else special = NAN;
}
if (d ≡ 0) special = (sign > 0) ? PZERO : NZERO;
```

This code is used in chunk 59.

66. In several functions we use integer values which have - in their binary representation - one single bit set.

⟨ global identifiers 58 ⟩ +≡

```
const integer integer_1 = integer(1);
const integer integer_52 = (integer_1 << 52) − integer_1;
const integer integer_32 = (integer_1 << 32) − integer_1;
const integer integer_20 = (integer_1 << 20) − integer_1;
```

67. The **todouble** function.

This function takes the given bigfloat and converts it into double format. If the bigfloat is in a special state (*NAN*, *PINF*, ...), we return the corresponding special double value. A bigfloat x is called *approximable* if

$$2^{-1074} \leq | x | < 2^{1024}.$$

For approximable values of x, we return the double nearest to x, otherwise bigfloats in special states *PINF*, *NINF*, *PZERO* or *NZERO* are returned.

In the main part of the function we distinguish the cases that the returned double is normalized or denormalized. At the end the significant s, the exponent *t_exp* and the significant *t_sig* of the double are put together.

⟨ general functions 14 ⟩ +≡

```
double todouble(const bigfloat &x)
{
  long s = ::sign(x.significant);
  long t_exp = 0;
  integer t_sig = 0;
  bigfloat rounded_value = abs(x);
  ⟨ special case checking of todouble 68 ⟩
  ⟨ rounding step of todouble 69 ⟩
  ⟨ check for normal or denormal return value 70 ⟩
  if (normal)
  ⟨ normal case 72 ⟩
  else ⟨ denormal case 71 ⟩
  double a;
  ⟨ set the bits of a 73 ⟩
  return a;
}
```

68. The special cases are first.

⟨ special case checking of todouble 68 ⟩ ≡

```
((bigfloat &) x).normalize();
if (isSpecial(x)) {
  if (ispZero(x)) return pZero_double;
  if (isnZero(x)) return nZero_double;
  if (isNaN(x)) return NaN_double;
```

$\quad$ **if** $(ispInf(x))$ **return** $pInf\_double$;
$\quad$ **if** $(isnInf(x))$ **return** $nInf\_double$;
}
This code is used in chunk 67.

69. We round the bigfloat such that all bits of the significant just fit into double format. We distinguish cases according to the quantity $log\_2 = exponent + precision$.

- If log_2 is between -1021 and 1024, we have to round to 53 places.
- If log_2 is between -1073 and -1022, we have to round to $1074 + log2$ places.

If log_2 is outside of $[-1073, 1023]$ the conversion overflows or underflows. Note that log_2 might change through the rounding.

$\langle$ rounding step of todouble 69 $\rangle \equiv$
integer $log\_2 = x.exponent + x.precision$;
if $(log\_2 > 1024)$ **return** $sign(x) * pInf\_double$;
if $(log\_2 < -1073)$ **return** $sign(x) * pZero\_double$;
if $(log\_2 > -1021)$ $rounded\_value = round(rounded\_value, 53, TO\_NEAREST)$;
else $rounded\_value = round(rounded\_value, 1074 + log\_2.tolong(\,), TO\_NEAREST)$;
This code is used in chunk 67.

70. Now we decide whether the bigfloat is approximable by a normalized double or not. This is the case if and only if

$$2^{-1022} \leq \mid rounded\_value \mid < 2^{1024}.$$

Note that log_2 can be 1025 because of rounding, in which case we return infinity.

$\langle$ check for normal or denormal return value 70 $\rangle \equiv$
long $normal = 1$;
$log\_2 = rounded\_value.exponent + rounded\_value.precision$;
if $(log\_2 \equiv 1025)$ **return** $sign(x) * pInf\_double$;
if $(log\_2 < -1021)$ $normal = 0$;
This code is used in chunk 67.

71. First we assume that we have a denormalized value to store. Remember that a denormalized double has an implicitly leading 0 bit and that its unbiased exponent is -1022. Let us consider the number k of zeros such that $rounded\_value = sig \cdot 2^e$ equals $0.\underbrace{00\ldots0}_{k}\ sig \cdot 2^{-1022}$. This is equivalent to $k = -1022 - e - l$ where l is the length of the significant sig of $rounded\_value$. We have to ensure that the bigfloat significant has length $52 - k$, that is, we shift the significant by $l\_shift = 52 - k - l$ places to the left. (If l_shift is negative rightshifts have to be performed). Simplifying we see that $l\_shift = e + 1074$.

$\langle$ denormal case 71 $\rangle \equiv$
{
$\quad$ **long** $l\_shift = rounded\_value.exponent.tolong(\,) + 1074$;

```
    if (l_shift > 0) t_sig = rounded_value.significant << l_shift;
    else t_sig = rounded_value.significant >> (-l_shift);
  }
```
This code is used in chunk 67.

72. Now we assume that our bigfloat $rounded\_value = sig \cdot 2^e$ is in normalized double range. As normalized doubles have an implicit leading one, the biased exponent of the returned double is

$$t\_exp = e + (l-1) + 1023$$

where again l is the length of sig. Finally we cut off the first bit of sig (which is necessarily one) and obtain the double significant t_sig.

⟨ normal case 72 ⟩ ≡
```
  {
    t_exp = rounded_value.exponent.tolong() + (rounded_value.precision - 1) + 1023;
      /* if the length of the rounded significant is lower than 53 we have to shift it to
       the left */
    t_sig = rounded_value.significant << (53 - rounded_value.significant.length());
    t_sig = t_sig & integer_52;
  }
```
This code is used in chunk 67.

73. Finally we set the bits of the returned double. That means that we have to lay down our specified values for sign, significant For this we use the function *compose_parts* and so we just need to calculate the arguments for this function.

⟨ set the bits of a 73 ⟩ ≡
```
  unsigned long sign, h_sig, l_sig;
  sign = (s == (-1));
  h_sig = ((t_sig >> 32) & integer_20).tolong();
  l_sig = (t_sig & integer_32).tolong();
  a = compose_parts(sign, t_exp, h_sig, l_sig);
```
This code is used in chunk 67.

74. Functions for input and output

There are two operators which perform input/output.

- The first one is the **operator**$\ll$. It outputs a bigfloat in either binary or decimal representation. The global variable *output_mode* determines the output form. The two different modes are
  - binary (*BIN_OUT*)
  - decimal (*DEC_OUT*)

  This operator uses the *binout* function for binary output and function *decimal_output* for decimal output.
- The second one is the operator $\gg$. It performs decimal input of a bigfloat using function *outofchar*. Binary input is not yet implemented.

We define the maximal size of a string field that is allocated during input and output.

⟨ global identifiers 58 ⟩ +≡

```
const long bin_maxlen = 10000;
```

75. Procedure *binout* takes an output stream and an integer b as input. It produces an unsigned, binary output of b.

⟨ functions for internal use 4 ⟩ +≡

```
void binout(ostream &os, integer b)
{
  char temp[bin_maxlen];
  long flag = 0, count = 0;
  if (b < 0) b *= −1;
  do {
    temp[count++] = (char) (b % 2).tolong() + '0';
    if (b ≤ 1) flag = 1;
    else b /= 2;
  } while (¬flag);
  for (long i = count − 1; i ≥ 0; i−−) os ≪ temp[i];
}
```

76. The following function computes the n^{th} power of a bigfloat in precision *prec* and rounding mode *mode*. [3]. We use it with $n = 10$ for our decimal output.

⟨ functions for internal use 4 ⟩ +≡

```
bigfloat powl(const bigfloat &x, long n, long prec = 1, rounding_modes
        mode = EXACT)
{
  bigfloat z = 1, y = x;
  long n_prefix = n;
```

[3] This does not mean that the result is correct up to *prec* digits. Only every operation within the procedure is carried out in that precision.

```
    while (n_prefix > 0) {
      if (n_prefix % 2) z = mul(z, y, prec, mode);
      n_prefix = n_prefix/2;
      y = mul(y, y, prec, mode);
    }
    return z;
  }
```

77. Procedure *decimal_output* displays a bigfloat b in decimal floating point notation, with *prec* decimal places in the significant. It first treats the sign of the given **bigfloat** b. The further procedure works with $\mid b \mid$ instead of b. Then we compute the decimal exponent of b which allows us to get the wanted *prec* decimal places of the output. Finally we write the output to **ostream** *os*.

⟨functions for internal use 4⟩ +≡

```
  void decimal_output(ostream &os, bigfloat b, long prec)
  {
    if (¬b.get_exponent( ).islong( ))
      error_handler(1, "decimal_output:␣not␣implemented␣for␣large␣exponents");
    if (prec ≡ 0)
      error_handler(1, "decimal_output:␣prec␣has␣to␣be␣bigger␣than␣0!");
    ostrstream oss;      /* string stream needed for output */
    long dd = 10;
    ⟨calculate the sign 78⟩
    ⟨compute decimal logarithm of b 79⟩
    ⟨compute decimal significant of b 80⟩
    ⟨output of the result 81⟩
  }
```

78. First we calculate the sign of b and take the modulus of b.

⟨calculate the sign 78⟩ ≡

```
  if (sign(b) < 0) { b = −b; os ≪ "-"; }
```

This code is used in chunk 77.

79. We need to know the decimal logarithm of b, rounded up to the next integer. For this we use the function *log10* from the standard math library that takes a double input. To avoid the problem of overflow in the double calculation we first determine $k = \lceil log_2(b) \rceil$, write $b = 2^k b_{rem}$ and finally compute $\log_{10}(b) = \log_{10}(2^k b_{rem}) = k \cdot \log_{10}(2) + \log_{10}(b_{rem})$.

⟨compute decimal logarithm of b 79⟩ ≡

```
  long log2_b = b.get_precision( ) + b.get_exponent( ).tolong( );
  bigfloat b_rem(b.get_significant( ), −b.get_precision( ));
  long log10_b = (long) ceil(log10(2) * log2_b + log10(todouble(b_rem)));
```

This code is used in chunk 77.

80. Now that we know that b has decimal length *log10_b* we proceed with the further preparation of the output. We have to ensure that the significant of the exponential output has exactly *prec* decimal digits. Therefor let $\mathit{diff} = \mathit{prec} - \mathit{log10\_b}$ and if *diff* is positive we multiply b with 10^{diff} and if it is negative we divide it by $10^{-\mathit{diff}}$. We use $\log_2 b + \mathit{log}_2 10^{\mathit{diff}}$ as precision for these operations. Afterwards the function *tointeger* is used to get a rounding to the next integer value.

⟨ compute decimal significant of b 80 ⟩ ≡
```
long diff = prec - log10_b;
long digits = (long) ceil(log2((double) dd) * diff) + log2_b;
bigfloat b_shift;
integer significant;
if (diff >= 0)
  b_shift = mul(b, powl(dd, diff, digits, TO_NEAREST), digits, TO_NEAREST);
else b_shift = div(b, powl(dd, -diff, digits, TO_NEAREST), digits, TO_NEAREST);
significant = tointeger(b_shift, TO_NEAREST);
oss << significant;
```
This code is used in chunk 77.

81. At last, we have to output the computed value. We take a floating point format with one digit in front of the decimal point. We cut off final zeros to get a clear output.

⟨ output of the result 81 ⟩ ≡
```
char *str = oss.str();
char *help = str + strlen(str) - 1;
while (*help == '0') help--;
*(help + 1) = 0;
if (prec > 1) os << (*(str++)) << "." << str;
else os << (*str);
if (log10_b != 1) os << "E" << log10_b - 1;
```
This code is used in chunk 77.

82. Now we are ready to implement the stream operators. We start with the output. In the case of decimal output we have to determine the decimal precision for the significant of the exponential output. Therefor we pass the decimal logarithm of the significant of b as *prec* to the function *decimal_output*.

⟨ input/output operators 82 ⟩ ≡
```
ostream &operator<<(ostream &os, const bigfloat &b)
{
  if (isSpecial(b)) {
    if (isNaN(b)) return os << "NaN";
    if (ispInf(b)) return os << "+Inf";
    if (isnInf(b)) return os << "-Inf";
    if (ispZero(b)) return os << "+0";
    if (isnZero(b)) return os << "-0";
  }
  int sign_b = sign(b.significant);
```

```
    if (bigfloat :: output_mode ≡ BIN_OUT) {
      if (sign_b < 0) os << "-";
      os << "0.";
      if (sign_b ≥ 0) binout(os, b.significant);
      else binout(os, −b.significant);
      os << "E";
      if (b.exponent < 0) os << "-";
      else os << "+";
      binout(os, b.exponent);
    }
    if (bigfloat :: output_mode ≡ DEC_OUT)
      decimal_output(os, b, max(1, (long) floor(log10(2) ∗ b.significant.length())));
    return os;
  }
```

See also chunk 83.
This code is used in chunk 10.

83. We turn to the input operator. The hard work is done by function *outofchar*.

```
⟨input/output operators 82⟩ +≡
  istream &operator>>(istream &is, bigfloat &b)
  {
    char temp[bin_maxlen];
    is >> temp;
    b = outofchar(temp);
    return is;
  }
```

84. Function *outofchar* provides the conversion of **string** to bigfloat. Currently, only decimal input is possible. The expected format is:

$$\pm dd\cdots d[.dd\cdots d[E \pm dd\cdots d]]$$

where d is out of $[0,9]$ and $\cdots$ stands for arbitrarily many d's. The procedure works as follows. We read the sign, the integer part, the decimal fraction and the exponent of the decimal floating point representation in turn. Then we concatenate the integer part and the decimal fraction into one decimal significant *sig* such that the input bigfloat is $sig \cdot 10^{exp-fl}$ where *fl* is the length of the decimal fraction and *exp* is the scanned exponent. In order to get a binary representation we either multiply or divide *sig* by 10^{exp-fl} according to the sign of $exp\_diff = exp - fl$. This operations are performed by the bigfloat operations *mul* and *div* respectively. It remains to specify an appropriate value for parameter *prec*. We know that the binary length of *sig* is l. In both cases the integer part of the result should be not larger than 10^{l+exp_diff}. The fraction should be smaller than 10^{fl}. Thus a precision of $\lceil \log_2 10(l + exp\_diff + fl)\rceil$ should be sufficient for both computations.

⟨functions for internal use 4⟩ +≡

```
bigfloat outofchar(char *rep, long prec = 0)
{
  integer sig = 0, exp = 0;
  bigfloat result, pow;
  long dd = 10;
  double log2dd = log2(dd);
  long int_length = 0, frac_length = 0;
  int s;
  ⟨scan sign s 85⟩
  int sign = s;
  ⟨scan integer part 87⟩
  ⟨scan fraction 88⟩
  ⟨scan exponent 89⟩
  long l = int_length + frac_length;
  long exp_diff = exp.tolong() − frac_length;
  if (prec ≤ 0) prec = (long) ceil(log2dd ∗ (l + exp_diff + frac_length));
  pow = powl(dd, abs(exp_diff), 1, EXACT);
  if (exp_diff > 0) result = mul(sig, pow, prec, TO_NEAREST);
  else result = div(sig, pow, prec, TO_NEAREST);
  if (sign ≡ 1) return result; else return −result;
}
```

85. First, we scan the optional sign at the beginning of the input.

⟨scan sign s 85⟩ ≡

```
s = 1;
if (rep[0] ≡ '-') { s = −1; rep++; }
else if (rep[0] ≡ '+') rep++;
```

This code is used in chunks 84 and 89.

86. We need a function *isnum* to test if a scanned character is a digit.

⟨auxiliary functions 16⟩ +≡

```
bool isnum(char ch)
{
  if ((ch ≥ '0') ∧ (ch ≤ '9')) return true;
  else return false;
}
```

87. Now we scan the input up to the decimal point if there is one. We pass every character and convert it to int. Each number is added to the temporary variable *sig* that is decimally shifted by one to the left.

⟨ scan integer part 87 ⟩ ≡

```
while (isnum(*rep)) {
   int_length++;
   sig = sig * dd + (*(rep++) - '0');
}
```

This code is cited in chunk 88.

This code is used in chunk 84.

88. The fraction scan works quite similar as ⟨ scan integer part 87 ⟩. We step over the input up to the character 'E' or until the input's end is reached.

⟨ scan fraction 88 ⟩ ≡

```
if (*rep ≡ '.') {
   rep++;
   while (isnum(*rep)) {
      sig = sig * dd + (*(rep++) - '0');
      frac_length++;
   }
}
```

This code is used in chunk 84.

89. To scan the exponent we first read the optional sign and otherwise proceed as before.

⟨ scan exponent 89 ⟩ ≡

```
if (*rep ≡ 'E') {
   rep++;
   ⟨ scan sign s 85 ⟩
   while (isnum(*rep)) exp = exp * dd + (*(rep++) - '0');
   if (s ≡ −1) exp = −exp;
}
```

This code is used in chunk 84.

90. References

References

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

[IE_Go] David Goldberg. What every computer scientist should know about floating-point arithmetic

[CACM95] K.Mehlhorn and S.Näher.LEDA: A library of efficient data types and algorithms.

[Näh95] S.Näher.LEDA manual.Technical report MPI-I-95-102, Max-Planck-Institut für Informatik, 1995

Index

List of Refinements