Moore

Interval Arithmetic in modern C++

by Walter F. Mascarenhas

Version 0.0.1, November, 2016

Contents

Chapter 1

The Moore library

1.1 Introduction

This document describes the Moore library, which implements interval arithmetic in modern C++. The purpose of this library is to allow users to use intervals in a simple, efficient and intuitive way. We would like to allow users to write code as simple as the next one in order to handle intervals with double precision endpoints.

```
Interval<> x("[1,2]");
Interval<> y(2,3);
Interval<> z = x + sin(y);
cout << "x + sin(y) = " << z << endl;
```

We also would like to to allow users more interested in speed than accuracy to write

```
Interval<float> x("[1,2]");
Interval<float> y(2,3);
Interval<float> z = x + sin(y);
cout << "x + sin(y) = " << z << endl;
```

and allow users looking for more accuracy to write something like

```
Interval<Real<1024>> x("[1,2]");
Interval<Real<1024>> y(2,3);
Interval<Real<1024>> z = x + sin(y);
cout << "x + sin(y) = " << z << endl;
```

where `Real<1024>` represents a floating point number with 1024 bits of mantissa.

Unfortunately, reality gets into our way and we must make compromises in order to achieve efficiency, consistency and easy of use. In the end, we do believe that we get close to these goals, and the users of the library will be able to write the following code

```
#include "moore/config/minimal.h" // the minimal declarations required to use the library:
....
Moore::RaiRounding r; // setting the rounding mode to upwards

Moore::Interval<> x("[1,2]");
Moore::Interval<> y(2.0f, 3.0f); // constants are floats, not ints.
Moore::Interval<> z = x + sin(y);
std::cout << "x + y = " << z << std::endl;
```

We can replace `Interval<>` in the code above by `Interval<T>`, where `T` is `float`, `double`, `long double`, `Moore::Quad` or `Moore::Real<N>`. The type `Moore::Quad` represents quadruple precision floating point numbers, and `Real<N>` handles floating point numbers with mantissa of `N` bits, for an integer `N >= 128` chosen at compile time. We can also write code which takes intervals with generic types of endpoints as arguments, as in:

```
template <End D, End E>
Hull<D,E> generic(Interval<D> const& x, Interval<E> const& y)
{
  auto h = exp(x + cos(y));
  auto w = x * sin(h) + y;
  return w + h;
}
```

Users of previous versions of the C++ language will not recognize the syntax

```
template <End D, End E>
Hull<D,E> generic(Interval<D> const& x, Interval<E> const& y)
```

used in the code above. They would be more used to

```
template <typename D, typename E>
Hull<D,E> generic(Interval<D> const& x, Interval<E> const& y)
```

and this is one point in which the Moore library differs from the previous C++ libraries for interval arithmetic. The Moore library was build using a new feature of the C++ language called "concepts", about which you can learn by searching with Google. In summary, concepts allow us to restrict the types used as parameters in templates, so that we get less spurious error messages due to failures in the compilation of templates. For instance, by using `End` instead of `typename` we ensure that only valid endpoints of intervals will be considered for the types `D` and `E`. They also help us to select which overloads of template function and classes should be used.

The key "concept" in the Moore library is `End`, which represents the endpoints of intervals. Using endpoints of type `E` we build intervals of type `Interval<E>`. We can operate with endpoints of different types `D` and `E`, and obtain endpoints of a type large enough to represent objects of both of them, which we call by `CommonEnd<D,E>`. Similarly, operating with intervals of type `Interval<D>` and `Interval<E>` we obtain an interval of type `Hull<D,E> = Interval<CommonEnd<D,E»`, which can represent exactly intervals of types `Interval<D>` and `Interval<E>`.

In summary, the main points in the library are:

- Endpoints: which are represented by the C++ concept `End` and are discussed in section 1.3.
- When we operate with endpoints of type `D` and `E` we obtain endpoints of type `CommonEnd<D,E>`.
- Intervals: for each type of endpoint `E` we have a corresponding interval class `Interval<E>`, and the family of all intervals is represented by the concept `CInterval`.
- Functions and operators: the library provides a comprehensive family of functions and operators to handle intervals and endpoints, which are listed in the Index of functions at the end of this document. It also implements most of the functions mandated by the IEEE standard for floating point arithmetic (see the Index of standard functions.) As we explain in Section 1.10, the library does not conform to the IEEE standard, but we do believe that it functions cover the most relevant part of it. It also has several useful functions which were not considered by the standard.
- When we operate with endpoints of types `D` and `E`, and the corresponding intervals `Interval<D>` and `Interval<E>`, we obtain intervals of type `Hull<D,E>`. If `E = D` then `Hull<D,E> = Interval<E>` and if only care about one type `E` of endpoint then you can read `Hull<D,E>` as `Interval<E>`.
- Endpoints and intervals are represented by the concept `EndOrInterval`, so that we can write code like the following function, which returns the convex hull of an arbitrary number of intervals and endpoints

  ```
  template <EndOrInterval... X>
  Hull<X...> hull(X const&... x){}

  auto i = hull(-1.0f, Interval<>(3.0,4.0), Interval<Quad>(3.0,4.0));
  ```

In the code above the compiler deduces the type of the interval `i` and we do not need to worry much about it. When you first use the Moore library, we suggest that you proceed in the same way: just write the code without paying to much attention to the technical details on which the library is based and see what happens. As you learn more about the library you will be able to do more sophisticated things, and we hope this document will help you to achieve your goals.

Ideally, you would be able to read the next section, learn how to compile and link the library and just write your code in an intuitive way, as in

```
template <End E>
void print_my_function(Interval<E> const\& x, Interval<E> const\& y)
{
  Interval<E> w = exp(x) * cos(y) + 4.0f * log(y);
  std::cout << w << std::endl;
}
```

As you try to write more advanced code it is likely that things will not work smoothly, and then you can read the rest of this document to learn more about the library. You may also read parts of the document in order to use more advanced options. For instance, you may not be pleased by the format in which the intervals are printed in the code above. In this case you should read Section 2.3 to learn about how to format the output.

1.2 How to use the library

The Moore library was written for people who know how to compile and link a C++ program. We assume that you use an IDE with which you can handle "c++ projects" with easy. For instance, the library was developed using QtCreator, which is a good open source IDE, and people using this IDE will probably have less trouble compiling and linking the library (and it will be easier for us to help them in case of trouble.)

Under the assumptions of the previous paragraph, you should follow these steps in order to use the library:

- You must use the compiler gcc 6.0, or a latter version of it. You should have the MPFR and GMP libraries installed in your machine. In order to use the quadruple precision type `Moore::Quad` you will also need the library quadmath.
- Send an email message to walter.mascarenhas@gmail.com asking for the Moore library. The subject of your message should contain the words "Moore library". We will appreciate if you tell us for what purpose you would like to use the library.
- We will send you a compressed file with the latest version of the library and of this document. Uncompress this file in a folder of your machine, which we will call `moore_root` from now on.
- Set the options of your project so that header files in the folder `moore_root/include` will be found by the compiler. For instance, if you use a makefile then it should contain something like

  ```
  INCPATH = -I /home/me/moore_root/include
  ```

  In QtCreator you could add an option like this one to your project:

  ```
  QMAKE_INCDIR += home/me/moore_root/include
  ```

- Make sure that you are using a version of C++ which supports concepts. In a makefile you would do something like

  ```
  CXXFLAGS = -std=c++1z -fconcepts ...
  ```

  In QtCreator you could add an option like this one to your .pro file:

  ```
  CONFIG+=c++1z
  CXXFLAGS+=-fconcepts
  ```

- Make sure that gcc's optimizer will handle properly the arithmetic operations used by the Moore library, by raising the following flags in your makefile:

  ```
  CXXFLAGS = -frounding-math -mfpmath=sse -msse2 -fsignaling-nans
  ```

  In QtCreator you could add options like this to your .pro file:

  ```
  CXXFLAGS+=-frounding-math CXXFLAGS+=-mfpmath=sse CXXFLAGS+=-msse2 CXXFLAGS+=-fsignaling-nans
  ```

- Link the MFPR and GMP libraries. In a makefile you would do something like

  ```
  LIBS = $(SUBLIBS) -lmpfr -lgmp
  ```

  In QtCreator you could add the following option to your .pro file:

  ```
  QMAKE_LIBS +=-lmpfr QMAKE_LIBS +=-lgmp
  ```

- Include all .cc files in the folder `moore_root/src/minimal` in your project. To use the quadruple precision `Quad` type then include also the file `moore_root/src/quad/quad.cc` and to use the type `Moore::Real<N>` include the file `moore_root/src/real/real.cc`.
- To use the `Moore::Quad` you must raise the `-fext-numeric-literals` compilation flag and link the `quadmath` library. In a makefile you would do something like

  ```
  CXXFLAGS = -fext-numeric-literals ...
  LIBS = $(SUBLIBS) -lquadmath -lmpfr -lgmp
  ```

  and in QtCreator you could add the following option to your .pro file:

  ```
  QMAKE_CXXFLAGS+=-fext-numeric-literals
  QMAKE_LIBS+= -lquadmath
  ```

- Choose a "compilation mode". At first, just choose the Debug mode on your IDE. As you learn more about the library, read Section 1.2.1 below.
- Once you are able to compile and link your code, be aware that you must set the rounding mode to upwards in order to use the library. You can achieve that by constructing an object of type `RaiiRounding`, which will restore the old rounding mode when it is destroyed. In summary, your code would look like this

  ```
  #include "moore/config/minimal.h"  // the basic declarations to use the library
  int main()
  {
    Moore::RaiiRounding r

    Your code goes here

    // implicit call to the destructor of r, resetting the rounding mode.
  }
  ```

  If you need to mix code using the library with code requiring other rounding modes then you should create objects of type Moore::RaiiRounding inside blocks, as in

  ```
  #include "moore/minimal.h"  // the basic declarations to use the library
  int main()
  {
    {
      Moore::RaiiRounding r
      Moore code here
      // implicit call to the destructor of r, resetting the rounding mode.
    }

    other code here

    {
      Moore::RaiiRounding r
      Moore code here
      // implicit call to the destructor of r, resetting the rounding mode.
    }
  }
  ```

1.2.1 Compilation modes

The Moore library can be compiled in three modes: Debug, Fast and Safe.

- In debug mode usually the optimizer is turned off and you get slower code, but with more warnings and the convenience of being able to follow your code step by step with a debugger. While you are learning about the library you should use this mode. In other to use it in an IDE you simply follow the usual procedure to generate code with debug information with this IDE. Formally, for the Moore library, you will be in Debug mode if `NDEBUG` is not defined when you compile your code. In Debug mode, the library uses assertions to check many things, and the debugger will stop your program and show you were the problem is in case an error is detected.

- In principle, the Fast mode will be turned on when you define `NDEBUG` (as most IDEs do when you ask for an optimized release build.) In this case, the Moore library does not check anything and bugs in your code may be undetected and lead to crashes, infinite loops or worse: incorrect results that look like as if they were correct. Therefore, you should only use the Fast mode when you are sure that your code is correct.
- Unfortunately, there are bugs which are introduced by the optimizer, or which only show up when `NDEBUG` is defined, and finding them is a painful process. In order to help you (and ourselves) to try to find such bugs, by defining both `NDEBUG` and `MOORE_IN_SAFE_MODE` you will compile the Moore library in Safe mode. In this mode, the library performs all the verifications it would perform in Debug mode, and signals an exception when a problem is found (see Section 1.8), and in the library's default configuration exceptions terminate the program. Even when your are confident that your code is correct, you may consider compiling it in Safe mode and running it just to check that everything is fine. Your "safe code" will be slower but you can let it run overnight for instance, just as a safety measure.

1.3 Endpoints

The main concept in the Moore library is called `End`, and represents the endpoints of the intervals. Formally, the library is based on few axioms about Endpoints, like:

A0 There is a set $\mathscr{E}$ of types of endpoints. For example, $\mathscr{E}$ could contain the types `float` and `double`.

A1 A type `E` is in $\mathscr{E}$ if and only if `is_end<E>()` returns true, and `E` satisfies the concept `End`. Such types are called endpoint types.

A2 The type `float` is in $\mathscr{E}$.

A5 An endpoint type contains at least one object `NAN` which represents not a numbers, in the IEEE 754 sense. It also contains an object to represent `+oo` and another to represent `-oo`.

A4 If `D` and `E` are in $\mathscr{E}$ then there exists a type `CommonEnd<D,E>` is $\mathscr{E}$, which can represent objects of type `D` and `E` exactly.

A5 `CommonEnd<float,E> = E`.

A6 If `D` and `E` are in $\mathscr{E}$ then the comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=` are defined for objects of type `D` and `E`, and return a boolean with meaning of the IEEE 754. In particular, `D::NAN == E::NAN` returns false, `D::NAN != E::NAN` returns true and `D::NAN < e` returns false for all `e` in `E`.

A7 Other technical details, like

```
CommonEnd<C,D,E> = CommonEnd<CommonEnd<C,D>,E>>.
```

However, axioms and formality are not the point of the library. We would like that to let you use it intuitively, thinking that there are 5 types of endpoints: `float`, `double`, `long double`, `Quad` and `Real<N>`. You can mix them in arithmetic operations and function calls and get reasonable results. In the next sections we look at each one of these types.

1.3.1 `float` is special

The type `float` receives an special treatment from the library. Intuitively, it is the smallest type that we support. In other words, we assume that there is an exact conversion of `floats` to all other endpoint types that we use. Of course, this restricts in the type of endpoints that which we can use with the library, but we believe that the trade off is positive. We can then simplify things and write (small) constants using `floats`. We can also express nan's and infinities simply as

```
NAN is the universal nan,   INFINITY is the universal positive infinity,
```

because since C++11 the macros `NAN` and `INFINITY` are part of the C++ standard and correspond to `float` constants with the meaning that we want.

As a result of the assumption above, for all endpoint types `E` supported by the library we can write code like

```
Interval<T> i(2.0f, INFINITY);
i = 0.5f * f + 2.0f;
```

1.3.2 `double`

We expect that `double` will be the most used type of endpoint. It combines efficiency with precision and this is the first type you should consider using. It is also the default endpoint type for the `Interval<T>` class, so that `Interval<>` represents an interval with `double` endpoints.

1.3.3 `long double`

`Long doubles` are more precise that `doubles`, but less efficient. As a result, code using `long doubles` is less used and less tested, and we do not know a single library using `long doubles` which does not have bugs on basic operations (or had bugs in previous versions.) Therefore, if you decide to use `long doubles` then you should test your code very carefully.

1.3.4 Quad

The type `Moore::Quad` is an alias to gcc's type `__float128`. It provides floating point arithmetic with mantissas of 113 bits. Unfortunately, due to a bug in the current implementation of `__float128`, `Quads` should not be used together with `long doubles`. The people responsible for implementing gcc's `__float128` say that this problem will be fixed in gcc's version 7.

Moreover, the square roots of `__float128` are not rounded sharply, and as a result the square roots of intervals with endpoints of this type may be slight larger than the tightest possible result.

1.3.5 Real<N>

The type `Moore::Real<N>` is a wrapper to the `__mpfr_struct` provided by the MPFR library. This wrapper is stack based (ie., the memory for the objects is allocated in he stack) and the number `N` of bits in the mantissa is fixed at compile time. Therefore, we do not offer the full power of the MPFR library, and you should not choose a huge value for `N`. For efficiency reasons, you should also pick `N` as a multiple of 64 (`N` must be at least 128.)

If you plan to use `Real<N>` in applications other than interval arithmetic then be aware that the arithmetic operations involving objects of this type are always rounded up.

1.3.6 CommonEnd

Given endpoint types `E1, E2, ..., EN` , the type `C = CommonEnd<Es...>` is such that objects of type `E1, E2, ..., EN` can be converted to `C` exactly. We also assume that the results of arithmetic operations of objects of type `EI` and `EJ` can be converted exactly to type `C`.

1.4 Intervals

There is only one type of interval in the Moore library: `Interval<E>`, which is parameterized by the endpoint type `E`. These intervals represent closed convex subsets of the real line, as

```
[] (the empty set),   [-oo,1],  [-2,3], [3,+oo].
```

The intervals are defined by two endpoints: its `inf` and `sup`, which may be finite on infinite (`inf` and `sup` are `NAN` for the empty interval.) For non empty intervals, we must have that `inf <= sup`, and the following entities are not intervals

```
[-oo,-oo],   [+oo,+oo],  [1,-1], [NAN,2]
```

Ideally, you should be able to use intervals intuitively: you can perform arithmetic operations with them, compute their `sin` and `cos` etc. The library offers the basic guarantee regarding these operations, ie., the computed value of a function `f` evaluated at an interval `[a,b]` contains the exact value `f(x)` for all `x in [a,b]`.

You can mix intervals with different types of endpoints, and usually an operation with intervals with endpoints of type `D` and `E` results in an interval of type `Hull<D,E> = Interval<CommonEnd<D,E> >`, which is described in the next section.

1.4.1 The Hull type

Given types `X1, X2, ..., XN` which are either interval or endpoint types, `H = Hull<Xs...>` is an interval type with endpoints of type `E` such that objects of all types `XI` with are endpoints can be converted exactly to `E`, and objects of all types `XI` with are intervals can be converted exactly to `H`. Moreover, the same exact conversions hold for the results of arithmetic operations involving objects of type `XI` and `XJ`.

1.5 Input and output

The intervals in the Moore library can be written and read from streams and converted to and from text. Interval can be read in the following ways:

```
Interval<E> x("[1,2]");          // constructing and interval from an string
Accuracy a;
Interval<E> y("[2,3]",a);        // now with information about the accuracy
bool v = try_parse(x,"[4,5"]);   // trying to parse an string
std::cin >> y;                   // reading an interval from a stream
try_scan(x, std::cin);           // trying to read one interval from a stream
std::vector<Interval<E>> is;
try_scan(is, std::cin);          // trying to read intervals from a stream
```

In the examples above, `try` functions return `true` or `false`, indicating whether the input is valid, and do not signal exceptions when the input is invalid. The other functions signal an exception when the input is invalid. By validity we mean that the input is a interval literal according to the IEEE standard for floating point arithmetic (see Section 1.9.)

Intervals can be written to streams and strings as in

```
Interval<E> i(1.0f);
std::vector<Interval<E>> is;
std::string format_string = "%IE%W10";
Format format_object(format_string);

std::string str = to_text(i);
str = to_text(i, format_string );

write(is, cout);
try_write(is, cout, format_object);

str = interval_to_exact(i);
write_exact(is, cout);

cout << format_object << i;
cout << Io::UPPER_CASE << i;
cout << std::setprecision(20) << std::right << i;
```

In these examples we use strings (like in printf) or an object of type `Io::Format` to specify the format of the output. In case of output streams, we can also use the standard manipulators. For instance, in the last line of the example we specify that the interval should be represented with at least 20 characters, and it should be right aligned within these 20 characters.

Besides the options provided by the usual C++ stream manipulators, the library provides many options for formatting the output. These options are listed in Tables 1.1 and 1.2. You can use a string of the form "%Option%Option%Option..." in functions which we take a format argument, or you can use an object `f` of type `Io::Format`, which you can construct from a string as above or calling functions of the form `set(f,option)`, and Table 1.2 contains a list of such options. You can also insert these options in the output stream, as in the example

```
Interval<E> i("[1.0,2.0]");
Io::Format f("%IA%AR%W20"); // Hexadecimal output, right aligned within 20 characters
write(cout, i, f);
os << Io::HEX << Io::RIGHT << Io::Width(20) << i; // the same as  the last line
```

Table 1.1: Options for formatting intervals (_ indicates blanks)

Name	Tag	Description	Samples		
Alignment	A	alignment within width	%AL (left), W9 [1,1]____	%AC) (center, W9 __[1,1]__	%AR (right, W9 ____[1,1]
Border slack	B	spaces between the border and the numbers	%B0 [1,1]	%B1 [_1,1_]	%B3 [___1,1___]
Center slack	,	spaces between the comma and the sup	%,0 [1,1]	%,1 [1,_1]	%,3 [1,___1]
Empty	Y	how empty is displayed	%YB blank []	%YI infs [+inf,-inf]	%YT text [empty]
Entire	E	how entire is displayed		%EI infs [-inf,+inf]	%ET [entire]
Exponent Width	X	spaces occupied by the exponent	%X1 [1.0e1,1.0e1]	%X2 [1.0e01,1.0e01]	%X3 [1.0e001,1.0e001]
Infinity	F	how infinity is displayed	%FS short [-inf,inf]	%FL = long, [infinity,infinity	
Inf Alignment	G	alignment of the inf	%GL left, %N3 [1__,1__]	%GC center, %N3 [_1_,1__]	%GR right, %N3 [__1,1__]
Number Width	N	space occupied by the numbers	%GR, %N3 [1__,1__]	%GI, %N3 [_1_,1__]	%GR, %N3 [__1,1__]
Notation	I	Equivalent to the a,f,g and e options in printf	%IF fixed [1.01, 2.02] %IE scientific [1.01e-1, 2.02e-1]	%IA, hexadecimal [0x1.0P+01,0x2.0P+01] %IG, shortest The shortest among %IF and %IE	
Padding	D	pad the output with zeros	%DN no [1,1]	%DY yes [1.00,1.00]	
Precision	P	digits after the decimal point	%P1 [1.1,1.2]	%P2 [1.11,1.18]	%P3 [1.111,1.178]
Overlap	O	Overlap style	%ON number [+1,+2]	%OT text [1,2]	
Show Sign	+	whether the + signs is shown	%+Y yes [+1,+2]	%+N no [1,2]	
Sup Alignment	S	alignment of the sup	S = left, N = 3 [1__,1__]	S = center, N = 3 [1__,_1_]	S = right, N = 3 [1__,__1]
Text case	C	case for the letters	%CL lower case [0x1p+1,0x1p+1]	%CU upper case, N = 3 [0X1P+1,0X1P+1]	S = right, N = 3 [1__,__1]
Width	W	space occupied by the interval	L = left, W = 9 [1,1]____	L = center, W = 7 _[1,1]_	L = right, W = 6 _[1,1]

1.6 Accuracy

The enum Accuracy indicates the accuracy with which strings are converted to intervals, or with which intervals are read from streams. Accuracy::Exact means that the endpoints of the interval were read exactly. Accuracy::Tight indicates that the read endpoints are with one ulp from the exact one encoded in the string or stream. Accuracy::Imprecise indicates that the endpoint read may differ from the exact one by many ulp's. Finally, Accuracy::Invalid indicates that the text in the string or stream does is not an interval literal.

```
enum class Accuracy
{
Empty = 0, Exact = 1, Tight = 2, Imprecise = 3, Invalid = 4
};
```

Table 1.2: Constants for formatting the outpup

Name	Option	Type	Description/Effect
CENTER	%AC	Align	Intervals are centered, as in _[1,1]_ Equivalent to std::ios_base::internal
DONT_PAD	%DN	Padding	Numbers are not padded with zeros
EMPTY_AS_BLANK	%YB	Empty	The empty interval is written as []
EMPTY_AS_INFS	%YI	Empty	The empty interval is written as [+inf,-inf]
EMPTY_AS_TEXT	%YT	Empty	The empty interval is written as empty
ENTIRE_AS_INFS	%EI	Entire	The entire interval is written as [-inf,+inf]
ENTIRE_AS_TEXT	%ET	Entire	The entire interval is written as entire
FIXED	%IF	Notation	Numbers are written in fixed notation
HEX	%IA	Notation	Numbers are written in hexadecimal format
INF_CENTER	%GC	InfAlign	The infimum in centered, as in [_1_,1]
INF_LEFT	%GL	InfAlign	The infimum in left aligned, as in [1__,1]
INF_RIGHT	%GR	InfAlign	The infimum in right aligned, as in [__1,1]
LEFT	%AL	Align	Intervals are left aligned, as in [1,1]___ Equivalent to std::ios_base::left
LONG_INFINITY	%FL	Infinity	Infinity is written as infinity
LOWER_CASE	%CL	TextCase	Output in lower case Equivalent to !std::ios_base::uppercase
NO_SIGN	%+N	Sign	The + sign is not shown Equivalent to !std::ios_base::showpos
OVERLAP_AS_NUMBER	%ON	OverlapStyle	Overlaps are written as numbers
OVERLAP_AS_TEXT	%OT	OverlapStyle	Overlaps are written as text
RIGHT	%AR	Align	Intervals are right aligned, as in __[1,1]. Equivalent to std::ios_base::right
SCIENTIFIC	%IE	Notation	Numbers are in scientific notation
SHORT	%IG	Notation	Numbers are written in the shortest among
SHORT_INFINITY	%FS	Infinity	Infinity is written as inf the fixed and scientific format
SUP_CENTER	%SC	SupAlign	The supremum in centered, as in [1,_1_]
SUP_LEFT	%SL	SupAlign	The supremum in left aligned, as in [1,1__]
SUP_RIGHT	%SR	SupAlign	The supremum in right aligned, as in [1,__1]
SIGN	%+Y	Sign	The + sign is shown Equivalent to std::ios_base::showpos
UPPER_CASE	%CU	TextCase	Output in upper case Equivalent to !std::ios_base::uppercase

1.7 Overlap

The enum Overlap has the 16 members indicated described in Figure 1.1, and an additional Undefined member. We use the following syntax to represent its members:

```
enum class Overlap : uint32_t
{
Undefined   = 0,       Overlaps    = 1 << 5,  Contains     = 1 << 11,
BothEmpty   = 1 << 0,  Starts      = 1 << 6,  StartedBy    = 1 << 12,
FirstEmpty  = 1 << 1,  ContainedBy = 1 << 7,  OverlappedBy = 1 << 13,
SecondEmpty = 1 << 2,  Finishes    = 1 << 8,  MetBy        = 1 << 14,
Before      = 1 << 3,  Equal       = 1 << 9,  After        = 1 << 15
Meets       = 1 << 4,  FinishedBy  = 1 << 10,
};
```

1.8 Exceptions

When the Moore library is compiled in Debug or Safe mode, exceptions lead to the termination of the program. In Safe mode, this termination is caused by a call to std::exit(EXIT_FAILURE) in the default error handler

provided by the library. This error handler also prints a message explaining the cause of the exception.

You can change this default behavior by linking a different error handler to your program. In order to that, you should write another version of the function `on_error` which is defined on the file `minimal/on_error.cc`, and link your own version of this file.

If the code is compiled in Fast mode then there is no checking for exceptions: we simply assume that you know that no exception will happen and would not like to pay any overheads. In this case, if your code is inconsistent then it may get into infinite loops, crash or worse: generate wrong results which look like correct ones.

1.9 Interval literals

The following table was copied from the IEEE standard 1788-2015 As in the IEEE standard, the following forms

Table 1.3: Grammar for literals, using the notation of 5.12.3 of IEEE Std 754-2008. Integer literal is integerLiteral, number literal is numberLiteral, interval literal is IntvlLiteral \t denotes the TAB character.

decDigit	[0123456789]
nonzeroDecDigit	[123456789]
hexDigit	[0123456789abcdef]
spaceChar	[\t]
natural	decDigit+
sign	[+-]
integerLiteral	{sign} ? {natural}
decSignicand	{decDigit} * ·̈{decDigit} + | {decDigit} + ·̈{decDigit} +
hexSignicand	{hexDigit} * "." {hexDigit} + | {hexDigit} + "." | {hexDigit} +
decNumLit	{sign} ? {decSignicand} ("e" {integerLiteral})?
hexNumLit	{sign} ? "0x" {hexSignicand} "p" {integerLiteral}
positiveNatural	("0")* {nonzeroDecDigit} {decDigit} *
ratNumLit	{integerLiteral} "/" {positiveNatural}
numberLiteral	{decNumLit} | {hexNumLit} | {ratNumLit}
sp	{spaceChar} *
dir	"d" | "u"
pointIntvl	"[" {sp} {numberLiteral} {sp} "]"
infSupIntvl	"[" {sp} {numberLiteral} {sp} "," {sp} {numberLiteral} {sp} "]"
radius	{natural}
uncertIntvl	{sign} ? {decSignicand} "?" {radius} ? {dir} ? ("e" {integerLiteral})?
IntvlLiteral	{pointIntvl} | {infSupIntvl} | {uncertIntvl}

of interval literal are also supported (the following lines were adapted from that standard):

- In the string `[l,u]`, the bound `l` may be `-oo` and `u` may be `+oo`. Any of `l` and `u` may be omitted, with implied values `l = -oo` and `u = +oo` respectively, e.g., `[,]` denotes Entire.

- The uncertain form with radius `?` is used for unbounded intervals, e.g., `m??d` denotes `[-oo,m]`, `m??u` `[m,+oo]` *m*??*u* denotes `[m, +oo]`, and `m??` denotes Entire with `m` being like a comment.

- The strings `[]` and `[ empty ]` whose value is Empty; the string `[ entire ]` whose value is Entire. Space between elements of a literal is optional: it denotes zero or more space characters. E.g., one may write `[empty]` or `[ empty ]` , etc.

1.10 Compliance with the IEEE Standard for interval arithmetic

The Moore is compliant to many aspects of the IEEE standard for arithmetic, but it does not comply to the standard in the following points:

- Decorations: The Moore library does not implement decorations, because we see no use for them in the use cases in which are most interested. Therefore, for us, decorations would introduce an overhead without any benefits.

- Exceptions: The treatment of exceptions in the Moore library depends on the compilation mode. In Debug mode exceptional cases cause the failure of assertions and the program is stopped by the debugger. In Safe mode exceptions are ignored, and if it is up to the user to make sure that exceptional situations do not occur. Finally, in Safe mode, exceptions are handled by an error handler, and the default error handler terminates the program.
- The sign of zero: The functions provided by the Moore library do not define the sign of their output when this output is equal to zero. In this case, it would be up to user to check whether the returned value has a positive or negative sign in this case.
- Several functions in the library differ in the way they treat empty sets from what is mandated by the standard. For instance, the functions `inf` and `sup` return `NAN` when their input is empty. Other functions, like `cancel_minus` and `cancel_plus` also handle infinite intervals differently from what is mandated by the standard. Therefore, you should read the documentation provided in the Reference chapter below in cases in which empty and infinite intervals are relevant for you.

1.11 Common mistakes

1.11.1 Explicit constructions

All constructors in the Moore Library are explicit, in order to avoid implicit (and unwanted conversions.) Unfortunately, this has the side effect that code like the following will not work:

```
Interval<> i = 1.0f; // Unfortunately, this is wrong
```

Instead, and you should write

```
Interval<> i(1.0f);
```

In our opinion, this was an unfortunate choice made the people responsible for the C++ language, and we believe that allowing implicit constructors would be worse than forbidding the invalid expression above.

1.11.2 Integer constants

The default type to express constants in the Moore library is `float`, and you should write

```
Interval<T> i(-1.0f);
Interval<T> j(-1.0f, 2.0f);
Interval<T> k = 0.5f * j + 2.0f;
```

instead of

```
Interval<T> i(-1);
Interval<T> j(-1, 2);
Interval<T> k = 0.5 * j + 2; // this will not work for \texttt{T = float}

Interval<T> i(-1.0f);
Interval<T> j(-1.0f, 2.0f);
Interval<T> k = 0.5f * j + 2.0f;
```

1.12 Reporting Bugs

As any software, it is likely that the Moore library contains bugs, and we ask the user to help us to fix them. If you do find something that looks like a bug we ask you to follow this procedure

- Read the Common Mistakes section and check whether this is indeed a bug or a "feature" of the library.
- Try to write the simplest code possible that shows the bug
- Send an email message to walter.mascarenhas@gmail.com with "Bug in the Moore library" as the subject. This message should contain a concise description of the bug and some code in which it occurs.

Table 10.7. The 16 states of interval overlapping situations for intervals a, b.
Notation $\forall_a$ means "for all a in $\boldsymbol{a}$," and so on. Phrases within a cell are joined by "and," e.g., `starts` is specified by $(\underline{a} = \underline{b} \wedge \overline{a} < \overline{b})$.

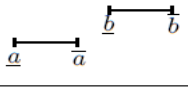
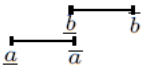
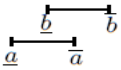
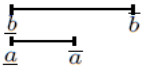
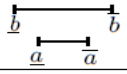
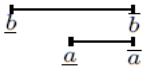
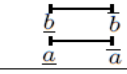
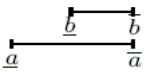
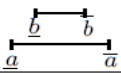
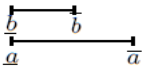
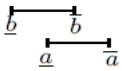
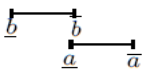
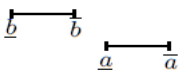
State $\boldsymbol{a} \circledcirc \boldsymbol{b}$ is	Set specification	Bound specification	Diagram
States with either interval empty			
`bothEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} = \emptyset$		
`firstEmpty`	$\boldsymbol{a} = \emptyset \wedge \boldsymbol{b} \neq \emptyset$		
`secondEmpty`	$\boldsymbol{a} \neq \emptyset \wedge \boldsymbol{b} = \emptyset$		
States with both intervals nonempty			
`before`	$\forall_a \forall_b\ a < b$	$\overline{a} < \underline{b}$	
`meets`	$\forall_a \forall_b\ a \leq b$ $\exists_a \forall_b\ a < b$ $\exists_a \exists_b\ a = b$	$\underline{a} < \overline{a}$ $\overline{a} = \underline{b}$ $\underline{b} < \overline{b}$	
`overlaps`	$\exists_a \forall_b\ a < b$ $\exists_b \forall_a\ a < b$ $\exists_a \exists_b\ b < a$	$\underline{a} < \underline{b}$ $\underline{b} < \overline{a}$ $\overline{a} < \overline{b}$	
`starts`	$\forall_b \exists_a\ a \leq b$ $\forall_a \exists_b\ b \leq a$ $\exists_b \forall_a\ a < b$	$\underline{a} = \underline{b}$ $\overline{a} < \overline{b}$	
`containedBy`	$\exists_b \forall_a\ b < a$ $\exists_b \forall_a\ a < b$	$\underline{b} < \underline{a}$ $\overline{a} < \overline{b}$	
`finishes`	$\exists_b \forall_a\ b < a$ $\forall_b \exists_a\ b \leq a$ $\forall_a \exists_b\ a \leq b$	$\underline{b} < \underline{a}$ $\overline{a} = \overline{b}$	
`equals`	$\forall_a \exists_b\ a = b$ $\forall_b \exists_a\ b = a$	$\underline{a} = \underline{b}$ $\overline{a} = \overline{b}$	
`finishedBy`	$\exists_a \forall_b\ a < b$ $\forall_a \exists_b\ a \leq b$ $\forall_b \exists_a\ b \leq a$	$\underline{a} < \underline{b}$ $\overline{b} = \overline{a}$	
`contains`	$\exists_a \forall_b\ a < b$ $\exists_a \forall_b\ b < a$	$\underline{a} < \underline{b}$ $\overline{b} < \overline{a}$	
`startedBy`	$\forall_a \exists_b\ b \leq a$ $\forall_b \exists_a\ a \leq b$ $\exists_a \forall_b\ b < a$	$\underline{b} = \underline{a}$ $\overline{b} < \overline{a}$	
`overlappedBy`	$\exists_b \forall_a\ b < a$ $\exists_a \forall_b\ b < a$ $\exists_b \exists_a\ a < b$	$\underline{b} < \underline{a}$ $\underline{a} < \overline{b}$ $\overline{b} < \overline{a}$	
`metBy`	$\forall_b \forall_a\ b \leq a$ $\exists_b \exists_a\ b = a$ $\exists_b \forall_a\ b < a$	$\underline{b} < \overline{b}$ $\overline{b} = \underline{a}$ $\underline{a} < \overline{a}$	
`after`	$\forall_b \forall_a\ b < a$	$\overline{b} < \underline{a}$	

Figure 1.1: Meaning of the values of the `Enum Overlap`. This page was copied from the IEEE Standard for floating point arithmetic, as indicated by the copyright above

Chapter 2

Reference

2.1 Arithmetic

`Hull<D,E> add(D const& x, Interval<E> const& y)` [Function]
`Hull<D,E> add(Interval<D> const& x, E const& y)` [Function]
`Hull<D,E> add(Interval<D> const& x, Interval<E> const& y)` [Function]
returns a tight interval containing the exact `x + y`.

`Hull<D,E> add_ends(D const& x, E const& y)` [Function]
Returns a tight interval containing the exact `x + y`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Hull<D,E> div(D const& x, Interval<E> const& y)` [Function]
`Hull<D,E> div(Interval<D> const& x, E const& y)` [Function]
`Hull<D,E> div(Interval<D> const& x, Interval<E> const& y)` [Function]
returns a tight interval containing the exact `x / y`.

`Hull<D,E> div_by_non_negative(Interval<D> const& x, Interval<E> const& y)` [Function]
returns a tight interval containing the exact `x / y`, under the assumption that `y` is not empty and is contained in `[0,+oo]`. If this assumption does not hold then an exception is signaled.

`Hull<D,E> div_by_positive(Interval<D> const& x, Interval<E> const& y)` [Function]
returns a tight interval containing the exact `x / y`, under the assumption that `y` is not empty and is contained in `(0,+oo]`. If this assumption does not hold then an exception is signaled.

`Hull<D,E> div_ends(D const& x, E const& y)` [Function]
returns a tight interval containing the exact `x / y`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Hull<C,D,E> fma(C const& x, D const& y, E const& z)` [Function]
`Hull<C,D,E> fma(C const& x, D const& y, Interval<E> const& z)` [Function]
`Hull<C,D,E> fma(C const& x, Interval<D> const& y, E const& z)` [Function]
`Hull<C,D,E> fma(C const& x, Interval<D> const& y, Interval<E> const& z)` [Function]
`Hull<C,D,E> fma(Interval<C> const& x, D const& y, E const& z)` [Function]
`Hull<C,D,E> fma(Interval<C> const& x, D const& y, Interval<E> const& z)` [Function]
`Hull<C,D,E> fma(Interval<C> const& x, Interval<D> const& y, E const& z)` [Function]
`Hull<C,D,E> fma(Interval<C> const&, Interval<D> const&, Interval<E> const&)` [Function]
returns a tight interval containing the exact `x * y + z`.

`Hull<D,E> minus_add_ends(D const& x, E const& y)` [Function]
returns a tight interval containing the exact `-(x + y)`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Hull<D,E> minus_div_ends(D const& x, E const& y)` [Function]
returns a tight interval containing the exact `-(x / y)`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Hull<D,E> minus_mul_ends(D const& x, E const& y)` [Function]

returns a tight interval containing the exact `-(x * y)`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Hull<D,E> mul(D const& x, Interval<E> const& y)` [Function]
`Hull<D,E> mul(Interval<D> const& x, E const& y)` [Function]
`Hull<D,E> mul(Interval<D> const& x, Interval<E> const& y)` [Function]

returns a tight interval containing the exact `x * y`.

`Hull<D,E> mul_ends(D const& x, E const& y)` [Function]

returns a tight interval containing the exact `x * y`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Interval<E> neg(Interval<E> const& i)` [Function]

returns minus the interval `i`.

`Hull<D,E> operator+(D const& x, Interval<E> const& y)` [Operator]
`Hull<D,E> operator+(Interval<D> const& x, E const& y)` [Operator]
`Hull<D,E> operator+(Interval<D> const& x, Interval<E> const& y)` [Operator]

returns a tight interval containing the exact `x + y`.

`Interval<D>& operator+=(Interval<D>& x, E const& y)` [Operator]
`Interval<D>& operator+=(Interval<D> const& x, Interval<E> const& y)` [Operator]

This operator is defined when `CommonEnd<D,E> = D`. It assign x + y to `x` and returns `x`.

`Hull<D,E> operator-(D const& x, Interval<E> const& y)` [Operator]
`Hull<D,E> operator-(Interval<D> const& x, E const& y)` [Operator]
`Hull<D,E> operator-(Interval<D> const& x, Interval<E> const& y)` [Operator]

returns a tight interval containing the exact `x - y`.

`Interval<D>& operator-=(Interval<D>& x, E const& y)` [Operator]
`Interval<D>& operator-=(Interval<D> const& x, Interval<E> const& y)` [Operator]

This operator is defined when `CommonEnd<D,E> = D`. It assign x - y to `x` and returns `x`.

`Hull<D,E> operator*(D const& x, Interval<E> const& y)` [Operator]
`Hull<D,E> operator*(Interval<D> const& x, E const& y)` [Operator]
`Hull<D,E> operator*(Interval<D> const& x, Interval<E> const& y)` [Operator]

returns a tight interval containing the exact `x * y`.

`Interval<D>& operator*=(Interval<D>& x, E const& y)` [Operator]
`Interval<D>& operator*=(Interval<D> const& x, Interval<E> const& y)` [Operator]

This operator is defined when `CommonEnd<D,E> = D`. It assign x * y to `x` and returns `x`.

`Hull<D,E> operator/(D const& x, Interval<E> const& y)` [Operator]
`Hull<D,E> operator/(Interval<D> const& x, E const& y)` [Operator]
`Hull<D,E> operator/(Interval<D> const& x, Interval<E> const& y)` [Operator]

returns a tight interval containing the exact `x / y`.

`Interval<D>& operator/=(Interval<D>& x, E const& y)` [Operator]
`Interval<D>& operator/=(Interval<D> const& x, Interval<E> const& y)` [Operator]

This operator is defined when `CommonEnd<D,E> = D`. It assign x / y to `x` and returns `x`.

`Interval<E> recip(Interval<E> const& i)` [Function]

returns a tight interval containing the exact `1.0f / i`.

`Interval<E> sqr(Interval<E> const& i)` [Function]

returns a tight interval containing the exact `i * i`.

`Interval<E> sqrt(Interval<E> const& i)` [Function]

For endpoint types `E` other than `Quad`, returns a tight interval containing the exact `sqrt(i)`. For `E = Quad` returns a interval which may be slightly larger than the tightest interval containing the exact `sqrt(i)` (due to a bug in the quadmath library.)

`Hull<D,E> sub(D const& x, Interval<E> const& y)` [Function]
`Hull<D,E> sub(Interval<D> const& x, E const& y)` [Function]
`Hull<D,E> sub(Interval<D> const& x, Interval<E> const& y)` [Function]

returns a tight interval containing the exact `x - y`.

`Hull<D,E> sub_ends(D const& x, E const& y)` [Function]

returns a tight interval containing the exact `x - y`. The function assumes that `x` and `y` are finite, and signal an exception otherwise.

`Interval<E> const& operator+(Interval<E> const& i)` [Operator]

returns `i`.

`Interval<E> operator-(Interval<E> const& i)` [Function]

returns minus the interval `i`.

2.2 Class Interval

2.2.1 Constructors for class Interval<E>

`Interval<E>()` [Constructor]

Creates an empty interval with endpoints of type `E`, as in

```
Interval<E> ie;      // creates an empty interval
ie = 1.0f;           // now ie = [1,1]
ie = Interval<E>();  // now ie becomes empty again
```

`Interval<E>(char const* str)` [Constructor]

If `str` is an interval literal, as described in Section **??**, then this constructs builds the corresponding interval. Otherwise it signals an exception. When the string is too complex this constructor may take a very long time. For extremely complex strings, it may also cause a memory allocation error, in which case an an exception is signaled.

`Interval<E>(char const* str, Accuracy& a)` [Constructor]

If `str` is an interval literal, as described in Section **??**, then this constructs builds the corresponding interval. Otherwise it signals an exception. On exit, the `Accuracy a` indicates how accurately the string was converted to the interval:

- `a = Accuracy::Empty` indicates that the string represents an empty interval
- `a = Accuracy::Exact` indicates that the conversion was exact
- `a = Accuracy::Tight` indicates that the sup was rounded to the nearest upward number of type `E` and inf was rounded to the nearest downward number of type `E`

When the string is too complex this constructor may take a very long time. For extremely complex strings, it may also cause a memory allocation error, in which case an an exception is signaled.

`Interval<E>(std::string const& str)` [Constructor]

The same as `Interval<E>(str.c_str())`.

`Interval<E>(std::string const& str, Accuracy& a)` [Constructor]

The same as `Interval<E>(str.c_str(),a)`.

`Interval<E>(D const& d)` [Constructor]

Requires that there is an exact conversion from endpoints of type `D` to endpoints of type `E` and constructs the interval `[d,d]` when `d` is a finite number, and signals an exception otherwise, as in

```
D d;
Interval<E> ie(d);         // constructs the interval [d,d].
Interval<E> ieb(NAN);      // error.
Interval<E> iec(INFINITY); // error.
```

`Interval<E>(C const& c, D const& d)` [Constructor]

Requires that there is an exact conversion from endpoints of type `C` and `D` to endpoints of type `E` and constructs the interval `[c,d]` when `c` and `d` are valid, and signals an exception in the cases point out as errors in the next example

```
Interval<E> i1(1.0f, 2.0f);           // constructs the interval [1,2].
Interval<E> i2(-INFINITY, 2.0f);      // constructs the interval [-oo,2].
Interval<E> i3(-INFINITY,  INFINITY); // constructs the interval [-oo,+oo].
Interval<E> i4(-INFINITY, -INFINITY); // Error, [-oo,-oo] is invalid.
Interval<E> i5( INFINITY,  INFINITY); // Error, [+oo,+oo] is invalid.
Interval<E> i6( NAN,       NAN);      // Error, invalid numbers
Interval<E> i7( NAN,       1.0f);     // Error, invalid numbers
Interval<E> i8( 1.0f,       NAN);     // Error, invalid numbers
Interval<E> i9( 2.0f,      1.0f);     // Error, numbers out of order
```

`Interval<E>(Interval<D> const& i)` [Constructor]

Requires that there is an exact conversion from endpoints `D` to endpoints `E` and constructs a copy of the interval `i`. This construction signals no exceptions.

2.2.2 Creators for class Interval<E>

By creator we mean an static method of the class `Interval<E>` which returns an interval.

`Interval<E> Interval<E>::after(D const& d)` [Creator]

This function assumes that there is an exact conversion from endpoints of type `D` to endpoints of type `E` and returns the interval `[d,+oo]`, signaling an exception when `d` is `NAN` or `+oo`.

`Interval<E> Interval<E>::after_m(D const& d)` [Creator]

This function assumes that there is an exact conversion from endpoints of type `D` to endpoints of type `E`, and returns the interval `[-d,+oo]`, signaling an exception when `d` is `NAN` or `-oo`. This function is more efficient than `after`.

`Interval<E> Interval<E>::after_zero()` [Creator]

Returns `[0,+oo]`.

`Interval<E> Interval<E>::after_zero(D const& d)` [Creator]

This function assumes that there is an exact conversion from endpoints of type `D` to endpoints of type `E` and returns the interval `[0,d]`, signaling an exception when `d` is `NAN` or negative.

`Interval<E> Interval<E>::before(D const& d)` [Creator]

This function assumes that there is an exact conversion from endpoints of type `D` to endpoints of type `E` and returns the interval `[-oo,d]`, signaling an exception when `d` is `NAN` or `-oo`.

`Interval<E> Interval<E>::before_zero()` [Creator]

Returns `[-oo,0]`.

`Interval<D> Interval<C>::center_and_radius(C const& c, R const& r)` [Creator]

This function assumes that the operations `c +/- r` are defined and that their result can be converted exactly to type `E`. It returns `[rounded_down(c - r), rounded_up(c + r)]`. An exception is signaled when `c` is not a finite number and when `e` is not a non negative number.

`Interval<E> Interval<E>::entire()` [Creator]

Returns the interval `[-oo,+oo]`.

`Interval<E> Interval<E>::raw(C const& c, D const& d)` [Creator]

This function is the most efficient way to create intervals. It is defined when endpoints of types `C` and `D` can be converted exactly to endpoints of type `E`. When `-c` and `d` define an interval then `[-c,d]` is returned (Note that there is a change in the sign of `c`, that is, `c` is MINUS the `inf` of the returned interval.) When `-c` and `d` do not define an interval then an exception is signaled.

```
Interval<E>::raw(2.0f, 1.0f);          // returns [-2,1]
Interval<E>::raw(NAN, NAN);            // returns empty, no exception is signaled
Interval<E>::raw(INFINITY, INFINITY);  // returns [-oo,+oo], no exception is signaled
Interval<E>::raw(INFINITY, 1.0f);      // returns [-oo,1]. no exception is signaled
Interval<E>::raw(-2.0f, 1.0f);         // an exception is signaled, [2,1] is invalid
Interval<E>::raw(NAN, 1.0f);           // an exception is signaled, [NAN,1] is invalid
Interval<E>::raw(-INFINITY, INFINITY); // an exception is signaled, [+oo,+oo] is invalid
Interval<E>::raw(INFINITY, -INFINITY); // an exception is signaled, [-oo,-oo] is invalid
```

`Interval<E> Interval<E>::raw_m(D const& d)` [Creator]

Creates the interval `[-d,-d]`. If `d` is `NAN` or infinite then a an exception is signaled.

`Interval<E> Interval<E>::round(Interval<D> const& id)` [Creator]

This function is defined when endpoints of type `D` can be rounded up and down to endpoints of type `E`. When texttti is empty it returns an empty interval. Otherwise it returns `[round_down(inf(i)), round_up(sup(i))]`, with the convention that

- `round_up(+oo) = +oo`,
- `round_up(-oo) = -oo`,
- `round_up(NAN) = NAN` and `empty = [NAN,NAN])`,
- `for finite d, round_up(d` is the least endpoint of type `E` which is greater than or equal to `d` (which may be `+oo`).
- for finite `d`, `round_downd` is the greatest endpoint of type `E` which is less than or equal to `d` (which may be `-oo`.)

`Interval<E> Interval<E>::symmetric(D const& d)` [Creator]

This function is defined when endpoints of type `D` can be converted exactly to endpoints of type `E`. If `d` is non negative, `Interval<E>::symmetric(d)` returns `[-d,d]`, otherwise it signals an exception.

`Interval<E> Interval<E>::zero()` [Creator]

Returns `[0,0]`.

2.3 Input and output

`void clear(Io::Format& f)` [Operator]

Assigns the following values to `f`'s attributes:

```
precision = 1, width = 0, border_slack = 0, center_slack = 1, number_width = 0,
exp_width = 1, alignment = left, empty = blank, entire = as text, infinity = short,
inf alignment = left, notation = shortest, padding = no, overlap = text,
show_sign = no, sup_alignment = left, text_case = lower_case
```

`Io::Align get_align(Io::Format const& f)` [Function]

`int get_border_slack(Io::Format const& f)` [Function]

`int get_center_slack(Io::Format const& f)` [Function]

`Io::Entire get_entire(Io::Format const& f)` [Function]

`Io::Empty get_empty(Io::Format const& f)` [Function]

`int get_exp_width(Io::Format const& f)` [Function]

`Io::InfAlign get_inf_align(Io::Format const& f)` [Function]

`Io::Infinity get_infinity(Io::Format const& f)` [Function]

`int get_number_width(Io::Format const& f)` [Function]

`Io::OverlapStyle get_overlap(Io::Format const& f)` [Function]

`Io::Padding get_padding(Io::Format const& f)` [Function]

`int get_precision(Io::Format const& f)` [Function]

`Io::SupAlign get_sup_align(Io::Format const& f)` [Function]

`Io::Sign get_sign(Io::Format const& f)` [Function]

`Io::TextCase get_text_case(Io::Format const& f)` [Function]

`int get_width(Io::Format const& f )` [Function]

gets the corresponding attribute of the `Format f`. See Table 1.1.

`std::string interval_to_exact(Interval<E> const& i)` [Operator]

Converts the interval `i` to a string `str` such that `try_parse(j, str)` would make `j` exactly equal to `i`.

`std::istream& operator»(std::istream& is, Interval<E>& i)` [Operator]

Tries to read the interval `i` from the input stream `is`. In case of failure an exception is signaled.

`std::ostream& operator«(std::ostream& os, Io::Align)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::BorderSlack)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::CenterSlack)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Entire)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Empty)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::ExpWidth)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::InfAlign)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Notation)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::NumberWidth)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::OverlapStyle)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Padding)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Precision)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::SupAlign)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Sign)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::TextCase)` [Operator]

`std::ostream& operator«(std::ostream& os, Io::Width)` [Operator]

Sets the corresponding attribute of the output `stream os`, as in Tables 1.1 and 1.2 and the examples below

```
cout << Io::ExpWidth(4); // setting the exponent width to 4 characters
cout << Io::HEX;         // setting the notation to hexadecimal
cout << Io::UPPER_CASE;  // setting the text case
cout << Io::RIGHT;       // right aligning text
cout << Io::Width(20);   // minimum of 20 characters per interval.
```

`std::ostream& operator«(std::ostream& os, Interval<E> const& ie)` [Operator]

Write the interval `id` from to the output stream `os`. The output can be formatted using the usual C++ stream manipulators and also by the several versions of the `operator«` mentioned below.

`Accuracy scan(Interval<E>& i, std::istream& is)` [Function]

`Accuracy scan(std::vector<Interval<E>& v, std::istream& is)` [Function]

The scan functions are similar to the `try_scan` functions, by they signal an exception in case of error.

`void set(Io::Format, Io::Align)` [Function]

`void set(Io::Format, Io::BorderSlack)` [Function]

`void set(Io::Format, Io::CenterSlack)` [Function]

`void set(Io::Format, Io::Entire)` [Function]

`void set(Io::Format, Io::Empty)` [Function]

`void set(Io::Format, Io::ExpWidth)` [Function]

`void set(Io::Format, Io::InfAlign)` [Function]

`void set(Io::Format, Io::Infinity)` [Function]

`void set(Io::Format, Io::MumberWidth)` [Function]

`void set(Io::Format, Io::OverlapStyle)` [Function]

`void set(Io::Format, Io::Padding)` [Function]

`void set(Io::Format, Io::Precision)` [Function]

`void set(Io::Format, Io::SupAlign)` [Function]

`void set(Io::Format, Io::Sign)` [Function]

`void set(Io::Format, Io::TextCase)` [Function]

`void set(Io::Format, Io::Width)` [Function]

Sets the corresponding attribute of the `Format f`. See Table 1.1.

`std::string to_text(Interval<E> const& i, Io::Format const& f)` [Function]

returns a string representing the interval according to the given format.

`std::string to_text(Interval<E> const& i)` [Function]

The same as `to_text(i, Io::Format("S"))`, which writes the interval in the short format.

`std::string to_text(Interval<E> const& i, char const* f)` [Function]

The same as `to_text(i, Io::Format(f))`.

`std::string to_text(Interval<E> const& i, std::string const& f` [Function]

The same as `to_text(i, Io::Format(f))`.

`Accuracy Interval<E> try_parse(Interval<E>& ie, char const* str)` [Function]

`Accuracy Interval<E> try_parse(Interval<E>& ie, std::string const& str)` [Function]

When str is not an interval literal as defined by the IEEE standards, the try_parse functions return Accuracy::Invalid. When str is an interval literal, e is set to the corresponding value, and the functions return an Accuracy which reflects how the string was interpreted in order to yield e. In realistic situations these functions cause no exceptions, but in principle, they may lead to program termination in case the input is so long or complex that it would lead the GMP library to cause a memory allocation error, and the Moore library takes no precaution to avoid such outlandish cases.

`bool Interval<E> try_parse(Io::Format& f, char const* str)` [Function]

`bool Interval<E> try_parse(Io::Format& f, std::string const& str)` [Function]

If the string is of the form `"%Option%Option...%Option"`, for the option in the table 1.1 then `f` then `f` is cleared, these new options are set and the function returns true. Otherwise, `f` is cleared and the function returns false

`Accuracy try_scan(Interval<E>& i, std::istream& is)` [Function]

Tries to read one interval from the input stream `is`. In case of failure returns `Accuracy::Invalid` and sets `is`'s fail bit. In case of success the returned value indicates the accuray with which the interval was read.

`Accuracy try_scan(std::vector<Interval<E>& v, std::istream& is)` [Function]

Tries to read all the intervals contained in the input stream `is`. In case of failure returns `Accuracy::Invalid` and sets `is`'s fail bit. In case of success the returned value indicates the accuracy with which the least accurate interval was read.

```
template <IntervalIterator It>
bool try_write(It begin, It end, std::ostream& os)                    [Function]
template <IntervalIterator It>
bool try_write(It begin, It end, std::ostream& os, char const* format)  [Function]
template <IntervalIterator It>
bool try_write(It begin, It end, std::ostream& os, std::string const& format) [Function]
template <IntervalIterator It>
bool try_write(It begin, It end, std::ostream& os, Io::Format const& format) [Function]
bool try_write(std::vector<Interval<E> const& v, std::ostream& os)    [Function]
bool
try_write(std::vector<Interval<E> const& v, ostream& os, char const* frmt)  [Function]
bool
try_write(std::vector<Interval<E> const& v, ostream& os, string const& frmt) [Function]
bool
try_write(std::vector<Interval<E> const& v, ostream& os, Format const& frmt) [Function]
```

Tries to write the vectors in the given range to the output stream `os`, with output formatted according to the format parameter (when the format is not provide the default value `Io::Format("S")` is used. The output can also be formatted using the `operator«` described above. Returns `true` if all intervals are written and `false` otherwise.

`template <IntervalIterator It>`
`void write(It begin, It end, std::ostream& os)` [Function]

`template <IntervalIterator It>`
`void write(It begin, It end, std::ostream& os, char const* format)` [Function]

`template <IntervalIterator It>`
`void write(It begin, It end, std::ostream& os, std::string const& format)` [Function]

`template <IntervalIterator It>`
`void write(It begin, It end, std::ostream& os, Io::Format const& format)` [Function]

`void write(std::vector<Interval<E> const& v, std::ostream& os)` [Function]

`void write(std::vector<Interval<E> const& v, ostream& os, char const* frmt)` [Function]

`void write(std::vector<Interval<E> const& v, ostream& os, string const& frmt)` [Function]

`void write(std::vector<Interval<E> const& v, ostream& os, Format const& frmt)` [Function]

The write functions are similar to the `try_write` functions, but they signal an exception in case of error.

2.4 Boolean functions of intervals

`bool are_disjoint(Interval<D> const& id, Interval<E> const& ie)` [Function]

Indicates whether the intervals `id` and `ie` are disjoint.

`Interval<E> has_negative(Interval<E> const& i)` [Function]

Indicates whether `i` contains a negative number.

`Interval<E> has_non_negative(Interval<E> const& i)` [Function]

Indicates whether `i` contains a non negative number.

`Interval<E> has_non_positive(Interval<E> const& i)` [Function]

Indicates whether `i` contains a non positive number.

`Interval<E> has_positive(Interval<E> const& i)` [Function]

Indicates whether `i` contains a positive number.

`Interval<E> has_zero(Interval<E> const& i)` [Function]

Indicates whether `0` is contained in `i`.

`bool intersect(Interval<D> const& id, Interval<E> const& ie)` [Function]

Indicates whether the intersection of the intervals `id` and `ie` is empty (this is the same as `!are_disjoint(id,ie)`.)

`bool is_bounded(Interval<E> const& i)` [Function]

Indicates whether `i` is bounded, ie., whether it is empty or its upper and lower bounds are finite.

`bool is_bounded_above(Interval<E> const& i)` [Function]

Indicates whether `i` is bounded above, ie., whether it is empty or its upper bound is finite.

`bool is_bounded_below(Interval<E> const& i)` [Function]

Indicates whether `i` is bounded below, ie., whether it is empty or its lower bound is finite.

`bool is_common_interval(Interval<E> const& i)` [Function]

Indicates whether `i` is not empty an has finite upper and lower bounds.

`bool is_empty(Interval<E> const& i)` [Function]

Returns true if and only if `i` is empty.

`bool is_entire(Interval<E> const& i)` [Function]

Returns true if and only if `i == [-oo,+oo]`.

`bool is_interior(D const& id, Interval<E> const& ie)` [Function]

Indicates whether the point `d` is contained in the interior of `ie`

`bool is_interior(Interval<D> const& id, Interval<E> const& ie)` [Function]

Indicates whether `id` is contained in the interior of `ie`

`bool is_less(Interval<D> const& x, Interval<E> const& y)` [Function]

Indicates whether `id` is weakly less than `ie`, in the sense that for all `d` in `id` there exists `e` in `ie` such that `d <= e` and for all `e` in `e` there exists `d` in `id` such that `d <= e`.

`bool is_member(E const& e, Interval<D> const& i)` [Function]

Indicates whether `e` is contained in `i`.

`bool is_negative(Interval<E> const& i)` [Function]

Indicates whether `i` is not empty and all its elements are negative.

`bool is_non_negative(Interval<E> const& i)` [Function]

Indicates whether `i` is not empty and all its elements are non negative.

`bool is_non_positive(Interval<E> const& i)` [Function]

Indicates whether `i` is not empty and all its elements are non positive.

`bool is_positive(Interval<E> const& i)` [Function]

Indicates whether `i` is not empty and all its elements are positive.

`bool is_singleton(Interval<E> const& i)` [Function]

Indicates whether `i` contains just one real number.

`bool is_strictly_less(Interval<D> const& x, Interval<E> const& y)` [Function]

Indicates whether `id` is strictly less than `ie`, in the sense that for all `d` in `id` there exists `e` in `ie` such that `d < e` and for all `e` in `e` there exists `d` in `id` such that `d < e`.

`bool is_subset(Interval<D> const& id, Interval<E> const& ie)` [Function]

Indicates whether `id` is contained in `ie`.

`bool is_valid(Interval<E> const& i)` [Function]

Returns true if and only if `i` is valid. This function is used for debug purposes only: except for bugs, the Moore library does not produce invalid intervals. In fact, if an invalid interval is detected the program should be interrupted immediately and we ask the user to send us a bug report.

`bool is_zero(Interval<E> const& i)` [Function]

Indicates whether `i == [0,0]`.

`Interval<E>& Interval<E>::operator=(Interval<D> const& id)` [Operator]

This function is defined when endpoints of type `D` can be converted exactly to endpoints of type `E`, and is used as in

```
Interval<E> ie;
Interval<D> id("[1,2]");
ie = id;                // now ie becomes [1,2].
```

`Interval<E>& Interval<E>::operator=(D const& d)` [Operator]

This function is defined when endpoints of type `D` can be converted exactly to endpoints of type `E`, and is used as in

```
D d(1.0f)
Interval<E> i;
i = d;        // now ie becomes [1,1].
```

`Interval<E>& Interval<E>::operator=(char const* str)` [Operator]

`Interval<E>& Interval<E>::operator=(std::string const& str)` [Operator]

When `str` is a valid interval literal, `operator=` assigns the corresponding interval to the object calling the operator. Otherwise, an exception is signaled, as in the following code:

```
Interval<E> i;
i = "[1,2];                  // now i becomes [1,2].
i = "I'm not an interval";   // an exception is signaled.
```

`bool operator==(Interval<D> const& id, Interval<E> const& ie)` [Operator]

Returns true if and only if the intervals `id` and `ie` are equal. In particular, two empty intervals are equal to each other and are different from all non empty intervals.

`bool operator==(Interval<D> const& i, E const& e)` [Operator]
`bool operator==(E const& e, Interval<D> const& i)` [Operator]

Returns true if and only if the intervals `i` is equal to `[e,e]`.

`bool operator!=(Interval<D> const& x, Interval<E> const& y)` [Operator]
`bool operator!=(Interval<D> const& x, E const& y)` [Operator]
`bool operator!=(E const& x, Interval<D> const& y)` [Operator]

In the three cases above `x != y` is equivalent to `!(x == y)`

`bool precedes(Interval<D> const& x, Interval<E> const& y)` [Function]

If either `id` or `id` is empty then this function returns true. Otherwise it indicates whether `sup(id) <= inf(ie)`.

`bool precedes(Interval<D> const& x, Interval<E> const& y)` [Function]

If either `id` or `id` is empty then this function returns true. Otherwise it indicates whether `sup(id) < inf(ie)`.

`Interval<E> zero_is_interior(Interval<E> const& i)` [Function]

Indicates whether `0` is contained in the interior `i`.

2.5 Interval functions of intervals

`Interval<E> acos(Interval<E> const& i)` [Function]

Returns a tight interval which contains `acos(x)` for all `x` in `i`.

`Interval<E> acosh(Interval<E> const& i)` [Function]

Returns a tight interval which contains `acosh(x)` for all `x` in `i`.

`Interval<E> asin(Interval<E> const& i)` [Function]

Returns a tight interval which contains `asin(x)` for all `x` in `i`.

`Interval<E> asinh(Interval<E> const& i)` [Function]

Returns a tight interval which contains `asinh(x)` for all `x` in `i`.

`Interval<E> atan(Interval<E> const& i)` [Function]

Returns a tight interval which contains `atan(x)` for all `x` in `i`.

`Interval<CommonEnd<D,E> > atan2(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval which contains `atan(d,e)` for all `d` in `id` and `e` in `ie`.

`Interval<E> atanh(Interval<E> const& i)` [Function]

Returns a tight interval which contains `atanh(x)` for all `x` in `i`.

`Interval<E> ceil(Interval<E> const& i)` [Function]

Returns a tight interval which contains `ceil(x)` for all `x` in `i`.

`Interval<E> cos(Interval<E> const& i)` [Function]

Returns a tight interval which contains `cos(x)` for all `x` in `i`.

`Interval<E> cosh(Interval<E> const& i)` [Function]

Returns a tight interval which contains `cosh(x)` for all `x` in `i`.

`Interval<E> exp(Interval<E> const& i)` [Function]

Returns a tight interval which contains `exp(x)` for all `x` in `i`.

`Interval<E> exp2(Interval<E> const& i)` [Function]
Returns a tight interval which contains `exp2(x)` for all `x` in `i`.

`Interval<E> exp10(Interval<E> const& i)` [Function]
Returns a tight interval which contains `exp10(x)` for all `x` in `i`.

`Interval<E> fabs(Interval<E> const& i)` [Function]
Returns the interval formed by `e` for `e` in texttti. Note that there is no function `abs` in the Moore library. You should use `fabs` instead.

`Interval<E> floorInterval<E> const& i)` [Function]
Returns a tight interval which contains `floor(x)` for all `x` in `i`.

`Interval<E> log(Interval<E> const& i)` [Function]
Returns a tight interval which contains `log(x)` for all `x` in `i`.

`Interval<E> log2(Interval<E> const& i)` [Function]
Returns a tight interval which contains `log2(x)` for all `x` in `i`.

`Interval<E> log10(Interval<E> const& i)` [Function]
Returns a tight interval which contains `log10(x)` for all `x` in `i`.

`Interval<E> max(D const& d, Interval<E> const& i)` [Function]
Returns a tight interval which contains `max(x,d)` for all `x` in `i`.

`Interval<E> max(Interval<D> const& i, E const& e)` [Function]
Returns a tight interval which contains `max(x,e)` for all `x` in `i`.

`Interval<E> max(Interval<D> const& id, Interval<E> const& ie)` [Function]
Returns a tight interval which contains `max(d,e)` for all `d` in `id` and `e` in `ie` and .

`Interval<E> min(D const& d, Interval<E> const& i)` [Function]
Returns a tight interval which contains `min(x,d)` for all `x` in `i`.

`Interval<E> min(Interval<D> const& i, E const& e)` [Function]
Returns a tight interval which contains `min(x,e)` for all `x` in `i`.

`Interval<E> min(Interval<D> const& id, Interval<E> const& ie)` [Function]
Returns a tight interval which contains `min(d,e)` for all `d` in `id` and `e` in `ie` and .

`Interval<E> negative_part(Interval<E> const& i)` [Function]
Returns the intersection of `i` with `[-oo,0]`.

`Interval<E> positive_part(Interval<E> const& i)` [Function]
Returns the intersection of `i` with `[0,+oo]`.

`Hull<D,E> pow(Interval<D> const& id, Interval<E> const& ie)` [Function]
Returns a tight interval which contains `d^e` for all `d > 0` in `id` and `e` in `ie`. If `ie` is not empty and `d` contains `0` then the output contains `0`.

`Interval<E> pown(Interval<E> const& i, int64_t n)` [Function]
Returns a tight interval which contains `x^p` for all `x` in `i`.

`Interval<E> round_ties_away(Interval<E> const& i)` [Function]
Returns a tight interval which contains `r(x)` for all `x` in `i`, where `r` is the function which rounds `x` to the closest representable integer, with ties broken away from zero.

`Interval<E> round_ties_to_even(Interval<E> const& i)` [Function]
Returns a tight interval which contains `r(x)` for all `x` in `i`, where `r` is the function which rounds `x` to the closest representable integer, with ties broken to even.

`Interval<E> sign(Interval<E> const& i)` [Function]

Returns a tight interval which contains `sign(x)` for all `x` in `i`, with `sign(x) = 1` for `x > 0`, with `sign(0) = 0` and with `sign(x) = -1` for `x < 0`.

`Interval<E> sin(Interval<E> const& i)` [Function]

Returns a tight interval which contains `sin(x)` for all `x` in `i`.

`Interval<E> tan(Interval<E> const& i)` [Function]

Returns a tight interval which contains `tan(x)` for all `x` in `i`.

`Interval<E> trunc(Interval<E> const& i)` [Function]

Returns a tight interval which contains `trunc(x)` for all `x` in `i`, where `trunc` is the function that truncates the fractional part of `x`.

2.6 Numeric functions of intervals

`CommonEnd<D,E> diff(Interval<D> const& id, Interval<E> const& ie)` [Function]

Measures the difference of `id` and `ie`. When one of the intervals is empty it returns the width of the other (and the width of empty is zero.) Otherwise, it returns `|sup(id) - sup(ie)| + |inf(id) - inf(ie)|`, evaluated rounding up in `CommonEnd<D,E>`'s arithmetic, with the convention that `+oo - +oo` $\bar{0}$ and `-oo - -oo = 0`.

`CommonEnd<D,E> dist(Interval<D> const& id, Interval<E> const& ie)` [Function]

When both intervals `id` and `ie` are not empty, `dist(id,ie)` returns the distance between them rounded up. When one of them is empty the function returns `NAN`.

`E inf(Interval<E> const& i)` [Function]

If `i` is empty then `inf` returns `NAN`. Otherwise, it returns the infimum of the interval as a set extended real numbers, that is, the smallest extended real number which is less than or equal to all `e` in `i`. In particular, `inf([entire]) = -oo`. As explained in Section **??**, when the returned value is zero its sign is undefined.

`E mag(Interval<E> const& i)` [Function]

When `i` is not empty `mag` returns the the supremum of `|e|` for `e` in `i`. When `i` is empty `mag` returns `NAN`.

`E mid(Interval<E> const& i)` [Function]

When `i` is bounded and not empty `mid` returns its midpoint rounded to nearest. The others cases are handled as follows, where `a` is finite number of type `E` and `M` is the largest finite element of `E`
`mid([]) = NAN, mid([-oo,+oo]) = 0, mid([-oo, a])= -M, mid([0,+oo)] = +M.`

`E mid_rough(Interval<E> const& i)` [Function]

Returns an approximation to `mid`. It assumes that `i` is not empty and that `d = |sup(i) + inf(i)|` is smallest than the largest finite number of type `E` (otherwise it signals an exception.) When `d` does not underflow, `mid_rough` differs from `mid` by at most one ulp. When `d` is subnormal `mid_rough` may differ form `mid` by at most the size of the smallest subnormal number. Note that in this case the absolute difference is tiny, but the relative difference may be of order 1.

`E mig(Interval<E> const& i)` [Function]

When `i` is not empty `mid` returns the distance of `i` to zero. When `i` is empty `mig` returns `NAN`.

`E minf(Interval<E> const& i )` [Function]

When `i` is not empty this function returns `-inf(i)`. When `i` is empty it returns `NAN`. This function is used for efficiency reasons. Due to the way in which intervals are represented internally by the Moore library, calling `minf(i)` is more efficient than calling `inf(i)`.

`E rad(Interval<E> const& i)` [Function]

If `i` is empty then `rad` returns `0`. Otherwise, it returns `(sup(i) - inf(i))/2` rounded up.

`E sup(Interval<E> const& i)` [Function]

If `i` is empty then `sup` returns `NAN`. Otherwise, it returns the supremum of the interval as a set extended real numbers, that is, the smallest extended real numbers which is greater than or equal to all `e` in `i`. In particular, `sup([entire]) = +oo`. As explained in Section **??**, when the returned value is zero its sign is undefined.

`E wid(Interval<E> const& i)` [Function]

If `i` is empty then `wid` returns `0`. Otherwise, it returns `(sup(i) - inf(i))` rounded up.

2.7 Reverse functions

`Hull<D,E> abs_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval containing the points `e` in `ie` such that `|e|` is in `id`.

`Hull<D,E> cancel_minus(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval containing the hull of the union of all intervals `ix` such that `ix + ie` is contained in `id`. Please note that this definitions may differ from the one used in the IEEE Standard for floating point arithmetic in cases in which the result is empty or entire.

`Hull<D,E> cancel_plus(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval containing the hull of the union of all intervals `ix` such that `ix - ie` is contained in `id`. Please note that this definitions may differ from the one used in the IEEE Standard for floating point arithmetic in cases in which the result is empty or entire.

`Hull<D,E> cosh_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns an interval containing the points `e` in `ie` such that `cos(e)` is in `id`. The returned interval is usually not much larger than the tightest interval possible. However, in rare cases the excess may of order one.

`Hull<D,E> cosh_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns an interval containing the points `e` in `ie` such that `cosh(e)` is in `id`. The returned interval may not be tight, but is usually reasonable.

`Hull<D,E> mul_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval containing the xs such that there exists `d` is in `id` for which `x * d` is in `ie`.

`Hull<D,E,X>`
`mul_rev(Interval<D> const& id, Interval<E> const& ie, Interval<X> const& ix)` [Function]

Returns an interval containing the points `x` in `ix` such that there exists `d` is in `id` for which `x * d` is in `ie`. This function usually returns a tight interval, but there are rare cases in which the tightest result would be empty but it returns an interval formed by a single point.

`std::pair<Hull<D,E>, Hull<D,E> >`
`mul_rev_to_pair(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a pair of tight intervals containing all points `x` such that there exists `d` is in `id` for which `x * d` is in `ie`.

`Interval<E> pown_rev(Interval<E> const& ie, int64_t p)` [Function]

Returns a tight interval containing the points `x` such that `x^p` is in `ie`.

`Hull<E,X> mul_rev(Interval<E> const& ie, , Interval<X> const& ix)` [Function]

Returns an interval containing the points `x` in `ix` such that `x^d` is in `ie`. This function usually returns a tight interval, but there are rare cases in which the tightest result would be empty but it returns an interval formed by a single point.

`Hull<D,E> sin_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns an interval containing the points `e` in `ie` such that `sin(e)` is in `id`. The returned interval is usually not much larger than the tightest interval possible. However, in rare cases the excess may of order one.

`Hull<D,E> sqr_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns a tight interval containing the points `e` in `ie` such that `x * x` is in `id`.

`Hull<D,E> tan_rev(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns an interval containing the points `e` in `ie` such that `tan(e)` is in `id`. The returned interval is usually not much larger than the tightest interval possible. However, in rare cases the excess may of order one.

2.8 Set operations

`Hull<X1,...XN> hull(X1 const& x1, ... XN const& xn)` [Function]

This function is defined when each type `XN` represents either an endpoint or an interval. It returns the convex hull of its arguments. The function signals an exception if one of the `xi` is `NAN` or an infinite endpoint.

```
template<RandomAccessIntervalIterator It>
typename std::iterator_traits<It>::value_type
hull(It begin, It end)
```
[Function]

The type `It` represents a random access iterator which iterates on intervals, and the function returns the convex hull of the range `[begin, end)`.

```
template<RandomAccessEndIterator It>
Interval< typename std::iterator_traits<It>::value_type >
hull(It begin, It end)
```
[Function]

The type `It` represents a random access iterator which iterates on endpoints, and the function returns the convex hull of the range `[begin, end)`. The function signals an exception if one of the elements of the range is `NAN` or infinite.

`Interval<E> hull(std::vector<Interval<E» const& v)` [Function]

Return the convex hull of the intervals in `v`, and `empty` when `v` is empty.

`Interval<E> hull(std::vector<E> const& v)` [Function]

Return the convex hull of the endpoints in `v`, and `empty` when `v` is empty. The function signals an exception if one of the elements of `v` is `NAN` or an infinite endpoint.

`Interval<E> hull(std::vector<Interval<E» const* v, int64_t n)` [Function]

When `n > 0`, this function returns the convex hull of the intervals `v[0],...,v[n-1]`, when `n = 0` it returns `[]` and when `n < 0` it it signals an exception.

`Interval<E> hull(E const* v, int64_t n)` [Function]

When `n > 0`, this function returns the convex hull of the endpoints `v[0],...,v[n-1]`, when `n = 0` it returns `[]` and when `n < 0` it it signals an exception. It also signals an exception if one of the elements of `v` is `NAN` or infinite.

`intersection<I1,...IN> hull(I1 const& i1, ... IN const& in)` [Function]

This function returns the intersection of its arguments.

```
template<IntervalIterator It>
typename std::iterator_traits<It>::value_type
intersection(It begin, It end)
```
[Function]

The type `It` represents an iterator which iterates on intervals, and the function returns the intersection of the range `[begin, end)`.

`Interval<E> intersection(std::vector<Interval<E» const& v)` [Function]

Return the intersection of the intervals in `v`, and `empty` when `v` is empty.

`Interval<E> intersection(std::vector<Interval<E» const* v, int64_t n)` [Function]

When `n > 0`, this function returns the intersection of the intervals `v[0],...,v[n-1]`, when `n = 0` it returns `[]` and when `n < 0` it it signals an exception.

`part_after<D,E> part_after(Interval<D> const& i, E const& e)` [Function]

Returns the intersection of `i` with `[e,+oo]`, when `e` is `NAN`. In particular, the function returns `i` when `e == -INFINITY` and `[]` when `e == INFINITY`.

`part_after<D,E> part_after(Interval<D> const& id, Interval<E> const& ie)` [Function]

Returns `id` when `ie` is empty and `part_after(id, sup(ie))` when `ie` is not empty.

`part_after<D,E> part_after_m(Interval<D> const& i, E const& e)` [Function]

Returns `part_after(i,-e)`, but is slightly more efficient.

`part_before<D,E> part_before(Interval<D> const& i, E const& e)` [Function]

Returns the intersection of `i` with `[-oo,e]`, when `e` is `NAN`. In particular, the function returns `i` when `e == +INFINITY` and `[]` when `e == -INFINITY`.

`part_after<D,E> part_before(Interval<D> const& id, Interval<E> const& ie)` [Function]
Returns `id` when `ie` is empty and `part_before(id, inf(ie))` when `ie` is not empty.

2.9 Miscellaneous functions

`Overlap overlap(Interval<D>& ie, Interval<E>& id)` [Function]
Returns a member of the enum `Overlap` indicating how `id` and `ie` are positioned with respect to each other.

`void swap(Interval<E>& x, Interval<E>& y)` [Function]
Swaps the intervals `x` and `y`.

Main index

Index of functions

Index of standard functions

Index of types