Directed Rounding Arithmetic Operations in C++

Guillaume Melquiond, Sylvain Pion

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Directed Rounding Arithmetic Operations in C++

Guillaume Melquiond — Sylvain Pion

N° 6757

December 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6757--FR+ENG

Directed Rounding Arithmetic Operations in C++

Guillaume Melquiond* , Sylvain Pion†

Thème SYM — Systèmes symboliques
Projets Geometrica et Proval

Rapport de recherche n° 6757 — December 2008 — 8 pages

Abstract: We propose the addition of new functions to the C++0x standard library that provide floating-point operations (`+`, `-`, `*`, `/`, `sqrt` and `fma`) as well as conversion functions with *directed rounding*. This set of functions is necessary to provide efficient support for interval arithmetic and related computations, and they directly map to IEEE-754 specifications. These functions require special compiler support due to their `constexpr` nature. This document is submitted to the ISO/WG21 working group which standardizes the C++ language, under the document N2811‡.

Key-words: interval arithmetic, directed rounding, floating-point arithmetic, C++, standardization, software design

* INRIA Saclay, France. Email: `guillaume.melquiond@inria.fr`
† INRIA Sophia-Antipolis, France. Email: `Sylvain.Pion@sophia.inria.fr`
‡ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2811.pdf

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Opérations arithmétiques avec arrondi dirigé en C++

Résumé : Nous proposons l'ajout de nouvelles fonctions à la bibliothèque standard de C++0x, qui fournissent des opérations concernant l'arithmétique flottante (`+`, `-`, `*`, `/`, `sqrt` et `fma`), ainsi que des fonctions de conversions avec un mode d'arrondi dirigé. Cet ensemble de fonctions est nécessaire pour fournir un support efficace à l'arithmétique d'intervalles et autres types de calculs similaires. Ces fonctions nécessitent un support particulier de la part du compilateur, à cause de leur nature `constexpr`. Ce document est soumis au groupe de travail ISO/WG21 qui standardise le language C++, sous la forme du document N2811[§].

Mots-clés : arithmétique d'intervalles, arrondis dirigés, arithmétique flottante, C++, standardisation, génie logiciel

[§] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2811.pdf

Contents

1 Motivation and context

WG21 has chosen to postpone the consideration of the proposal to add interval arithmetic to the standard library, N2137[1], after C++0x. While a new standard is being developed to provide a language-independent base for interval arithmetic (IEEE-1788), we believe that progress can be made to enhance low-level support for interval arithmetic in C++0x, and make it easier to build an interval arithmetic library on top of C++0x efficiently and portably.

The current CD provides the `fegetround` and `fesetround` functions in the <cfenv> header to access and change the current rounding mode. Those functions were imported from C99, and they are at the root of every interval arithmetic implementation. There are however two problems which prevent efficient support for interval arithmetic:

- the CD does not provide the accompanying `FENV_ACCESS` pragma, which instructs the compiler that a block of code cares about the rounding mode. The consequence is that one must use temporary volatile variables, or any other equivalent implementation-dependent method, which is inefficient. This can be considered as an incomplete addition to C++0x compared to C++03.
- building interval constants is very useful, but it is not possible to build `constexpr` interval operations since the restricted form of expressions allowed by `constexpr` functions forbids calling `fesetround`. Extending a general feature such as `constexpr` to support this, does not seem appropriate for this particular use.

2 A different approach

We propose to go away from `fesetround` as the low-level primitive used to implement interval arithmetic. Instead of a global state rounding mode, we propose to add functions with directed rounding modes that perform floating-point addition, subtraction, multiplication, division, square root and fma, as well as conversions between floating-point and integral types, and between floating-point types themselves. These functions correspond to the functionality mandated by IEEE-754-1985 (and by IEEE-754-2008 for fma).

These functions, especially as they would be `constexpr`, require specific compiler support.

On hardwares that support a static rounding direction in the floating-point operation instructions or that support several floating-point control registers, the new functions make it easier for the compiler to generate those instructions than currently, since using the `fesetround` interface around a block of floating-point operations requires some (possibly inter-procedural) flow analysis.

Fundamentally, there is no reason why C++0x provides built-in operators `+`, `-`, `*`, `/` which allow to build `constexpr` functions with the default rounding mode, while it would not be possible with its directed rounding cousins.

[1] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2137.pdf`

Note on optimization: on hardware that needs to set a register holding the rounding mode, using `fesetround` in a block of code can be more efficient. In practice however, this kind of computation is typically needed for adding vectors of intervals, and it is our belief that, in our approach, a quality compiler should be able to move the rounding mode change instructions outside of the loops.

2.1 Syntax and design choices

There are 4 rounding modes specified by IEEE-754-1985: to the nearest (default), towards zero, towards plus infinity and towards minus infinity.

We propose that the new functions use a constant rounding mode. That is, it would not be possible to pass it as an argument, and even less be a run time entity.

Moreover, C++0x currently has 2 ways of refering to rounding modes:

- `fesetround` takes an argument of type `int` with possible values `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO` and `FE_UPWARD`.
- the `floating_round_style` enum which has 4 values plus a fifth `round_indeterminate`.

We prefer the latter alternative for type checking and consistency reasons. Moreover, we suggest to pass the rounding mode as explicit template argument, to make sure it is a constant, while still not hardcoding it in the function name like `add_up` or `mul_down`. Passing `round_indeterminate` is not useful, and is not allowed.

An example of use would then be:

```
double d = add<round_toward_infinity>(a, b);
float pi_up = rounded_cast<round_toward_infinity, float>(3.14159265358979323);
int i = rounded_cast<round_toward_infinity, int>(134675./4247.);
```

Side note: some languages which support more mathematical symbols for operators, like Fortress[2], provide a nice syntax like `a` ⧋ `b`.

2.2 Promotion rules

Ideally, the `add` function should behave just like the built-in floating-point `operator+`, concerning the types of the operands, following the promotion rules. And similarly for the other functions. We propose to use constrained templates, `auto` and `decltype` to achieve this.

2.3 Side effects

Division by zero, or taking the square root of a negative number, can produce side effects like raising floating-point exceptions and setting `errno`. Obviously, this cannot work for

[2] `http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`

`constexpr` functions at compile-time. We therefore propose that the evaluation of these functions follow the IEEE-754 "default non-stop exception handling" mode (producing NaNs and Inf, but no floating-point exception nor `errno` change), and with the caveat that the compile-time evaluation is done without raising the corresponding status flags.

A compiler may choose to issue or not a warning in such cases.

2.4 Header file

The closest related headers are <`cmath`> and <`numeric`>. However, neither seem perfectly appropriate to provide these functions. Moreover, since these functions are compiler-supported, it makes sense to put them in a separate header file, so as to ease the work of library vendors. This header could then be provided by compiler vendors.

We therefore propose a new header file <`rounded_math`>, and a new section to describe it.

3 Acknowledgements

We thank Marc Glisse for his useful comments.

4 Proposal

26.8 Directed rounding functions

26.8.1 Header <rounded_math> synopsis

```
namespace std {
  template < float_round_style r, FloatingPointLike T, ArithmeticLike U >
  requires True<(r != round_indeterminate)>
  constexpr T rounded_cast(U u);

  template < float_round_style r, IntegralLike T, FloatingPointLike U >
  requires True<(r != round_indeterminate)>
  constexpr T rounded_cast(U u);

  template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
  requires True<(r != round_indeterminate)>
  constexpr auto add(T t, U u) -> decltype(t + u);

  template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
  requires True<(r != round_indeterminate)>
  constexpr auto sub(T t, U u) -> decltype(t - u);

  template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
  requires True<(r != round_indeterminate)>
  constexpr auto mul(T t, U u) -> decltype(t * u);

  template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
  requires True<(r != round_indeterminate)>
  constexpr auto div(T t, U u) -> decltype(t / u);

  template < float_round_style r, FloatingPointLike T >
  requires True<(r != round_indeterminate)>
  constexpr T sqrt(T t);

  template < float_round_style r, FloatingPointLike T, FloatingPointLike U,
             FloatingPointLike V >
  requires True<(r != round_indeterminate)>
  constexpr auto fma(T t, U u, V v) -> decltype(fma(t, u, v));
}
```

1 The compile-time evaluation of these functions ignores any floating-point status flags raised, if any.

2 The run-time evaluation of these functions shall not raise any floating-point exception, nor change `errno`.

```
template < float_round_style r, FloatingPointLike T, ArithmeticLike U >
requires True<(r != round_indeterminate)>
constexpr T rounded_cast(U u);

template < float_round_style r, IntegralLike T, FloatingPointLike U >
requires True<(r != round_indeterminate)>
constexpr T rounded_cast(U u);
```

Returns: The conversion `u` to the type `T` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
requires True<(r != round_indeterminate)>
constexpr auto add(T t, U u) -> decltype(t + u);
```

Returns: The addition of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
requires True<(r != round_indeterminate)>
constexpr auto sub(T t, U u) -> decltype(t - u);
```

Returns: The subtraction of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
requires True<(r != round_indeterminate)>
constexpr auto mul(T t, U u) -> decltype(t * u);
```

Returns: The multiplication of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T, FloatingPointLike U >
requires True<(r != round_indeterminate)>
constexpr auto div(T t, U u) -> decltype(t / u);
```

Returns: The division of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T >
requires True<(r != round_indeterminate)>
constexpr T sqrt(T t);
```

Returns: The square root of `t` rounded according to `r`.

```
template < float_round_style r, FloatingPointLike T, FloatingPointLike U,
           FloatingPointLike V >
requires True<(r != round_indeterminate)>
constexpr auto fma(T t, U u, V v) -> decltype(fma(t, u, v));
```

Returns: The fused multiply-and-add of `t`, `u`, and `v`, rounded according to `r`.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399