Complex Arithmetic Edits—Update

WG14/N620 X3J11/96-84 (Draft 10/15/96)

Jim Thomas
517 Hendon Court
Sunnyvale, CA 94087
jthomas@best.com

The changes in this update are:

1. The words "and exception behavior" were added in three places to clarify that the exception behavior as well as result is determined by the table.

2. The sample multiplication implementation was updated to catch overflow cases which the previous version allowed to result in NaN+iNaN instead of an infinity.

3. The sample division implementation was updated to catch non-zero/zero cases which the previous version allowed to result in NaN+iNaN instead of an infinity.

Y.5 Binary operators

The following subclauses supplement 6.3 in order to specify the type of the result for an operation with an imaginary operand.

For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow (7.X.1); in these cases the result satisfies certain properties (specified in Y.5.1), but is not completely determined.

Y.5.1 Multiplicative operators

Semantics

If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has complex type. (If either operand has complex type, then the result has complex type.)

If the operands are not both complex, then the result and exception behavior of the `*` operator is defined by the usual mathematical formula:

*	real *x*	imaginary *y*I*	complex *x + y*I*
real *u*	*x*u*	*(y*u)*I*	*(x*u) + (y*u)*I*
imaginary *v*I*	*(x*v)*I*	*-y*v + 0*I*	*(-y*v) + (x*v)*I*
complex *u + v*I*	*(x*u) + (x*v)*I*	*(-y*v) + (y*u)*I*	

If the second operand is not complex, then the result and exception behavior of the `/` operator is defined by the usual mathematical formula:

/	*x*	*y*I*	*x + y*I*
u	*x/u*	*(y/u)*I*	*(x/u) + (y/u)*I*
*v*I*	*(-x/v)*I*	*y/v + 0*I*	*(y/v) + (-x/v)*I*

A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a NaN). A complex or imaginary value is a *finite number* if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a *zero* if each of its parts is a zero. The `*` and `/` operators satisfy the following infinity properties for all real, imaginary, and complex operands[1]:

- If one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the `*` operator is an infinity.
- If the first operand is an infinity and the second operand is a finite number, then the result of the `/` operator is an infinity.
- If the first operand is a finite number and the second operand is an infinity, then the result of the `/` operator is a zero.
- If the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the `/` operator is an infinity.

If both operands of the `*` operator are complex or if the second operand of the `/` operator is complex, the operator raises exceptions if appropriate for the calculation of the parts of the result, and may raise spurious exceptions.

Examples

1. Multiplication of **`double complex`** operands could be implemented as follows. Note that the imaginary unit **`I`** has imaginary type (see Y.6).

```
#include <math.h>
#include <complex.h>
#define isinf(x) (fabs(x)==INFINITY)
```

[1] These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for **`CX_LIMITED_RANGE`** is *off*).

```
/* Multiply z * w ... */
double complex _Cmultd(double complex z, double complex w)
{
      FP_CONTRACT_OFF;
      double a, b, c, d, ac, bd, ad, bc, x, y;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      ac = a * c;  bd = b * d;
      ad = a * d;  bc = b * c;
      x = ac - bd;
      y = ad + bc;
      /* Recover infinities that computed as NaN+iNaN ... */
      if (isnan(x) && isnan(y))
      {
         int recalc = 0;
         if (isinf(a) || isinf(b))  /* z is infinite */
         {
            /* "Box" the infinity ... */
            a = copysign(isinf(a) ? 1.0 : 0.0, a);
            b = copysign(isinf(b) ? 1.0 : 0.0, b);
            /* Change NaNs in the other factor to 0 ... */
            if (isnan(c)) c = copysign(0.0, c);
            if (isnan(d)) d = copysign(0.0, d);
            recalc = 1;
         }
         if (isinf(c) || isinf(d))  /* w is infinite */
         {
            /* "Box" the infinity ... */
            c = copysign(isinf(c) ? 1.0 : 0.0, c);
            d = copysign(isinf(d) ? 1.0 : 0.0, d);
            /* Change NaNs in the other factor to 0 ... */
            if (isnan(a)) a = copysign(0.0, a);
            if (isnan(b)) b = copysign(0.0, b);
            recalc = 1;
         }
         if (!recalc)
         {
            /* Recover infinities from overflow cases ... */
            if (isinf(ac) || isinf(bd) || isinf(ad) || isinf(bc))
            {
               /* Change all NaNs to 0 ... */
               if (isnan(a)) a = copysign(0.0, a);
               if (isnan(b)) b = copysign(0.0, b);
               if (isnan(c)) c = copysign(0.0, c);
               if (isnan(d)) d = copysign(0.0, d);
               recalc = 1;
            }
         }
         if (recalc)
         {
            x = INFINITY * (a * c - b * d);
            y = INFINITY * (a * d + b * c);
         }
      }
      return x + I * y;
}
```

In ordinary (finite) cases, the cost to satisfy the infinity property for the **`*`** operator is only one **`isnan`** test. This implementation opts for performance over guarding against undue overflow and underflow.

2. Division of two **`double complex`** operands could be implemented as follows.

```
#include <math.h>
#include <complex.h>
/* isinf is as in example 1 above */

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
      FP_CONTRACT_OFF;
      double a, b, c, d, logbw, denom, x, y;
      long llogbw = 0;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      logbw = logb(fmax(fabs(c), fabs(d)));
      if (isfinite(logbw))
      {
         llogbw = (long)logbw;
         c = scalb(c, -llogbw);
         d = scalb(d, -llogbw);
      }
      denom = c * c + d * d;
      x = scalb((a * c + b * d) / denom, -llogbw);
      y = scalb((b * c - a * d) / denom, -llogbw);
      /*
       * Recover infinities and zeros that computed as NaN+iNaN;
       * the only cases are infinite/finite, finite/infinite,
       * and non-zero/zero ...
       */
      if (isnan(x) && isnan(y))
      {
         if ((isinf(a) || isinf(b)) && isfinite(c) && isfinite(d))
         {
            a = copysign(isinf(a) ? 1.0 : 0.0, a);
            b = copysign(isinf(b) ? 1.0 : 0.0, b);
            x = INFINITY * (a * c + b * d);
            y = INFINITY * (b * c - a * d);
         }
         else if ((logbw == INFINITY) && isfinite(a) && isfinite(b))
         {
            c = copysign(isinf(c) ? 1.0 : 0.0, c);
            d = copysign(isinf(d) ? 1.0 : 0.0, d);
            x = 0.0 * (a * c + b * d);
            y = 0.0 * (b * c - a * d);
         }
         else if ((denom == 0.0) && (isfinite(a) || isfinite(b)))
         {
            x = copysign(INFINITY, c) * a;
            y = copysign(INFINITY, c) * b;
         }
      }
      return x + I * y;
}
```

Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the `scalb` function, instead of with division, provides better roundoff characteristics.

Y.5.2 Additive operators

Semantics

If one operand has real type and the other operand has imaginary type, then the result has complex type. If both operands have imaginary type, then the result has imaginary type. (If either operand has complex type, then the result has complex type.)

In all cases the result and exception behavior of a **+** or **–** operator is defined by the usual mathematical formula:

$\pm$	x	$y*I$	$x + y*I$
u	$x \pm u$	$\pm u + y*I$	$(x \pm u) + y*I$
$v*I$	$x \pm v*I$	$(y \pm v)*I$	$x + (y \pm v)*I$
$u + v*I$	$(x \pm u) \pm v*I$	$\pm u + (y \pm v)*I$	$(x \pm u) + (y \pm v)*I$