Implementing Ada Fixed-point Types having Arbitrary Scales

Paul N. Hilfinger
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June, 1990

Abstract

Ada implementations have the option of supporting fixed-point arithmetic with arbitrary scale—that is, in which the safe numbers are of the form $m\sigma$ for a fixed, arbitrary rational $\sigma > 0$, with m in some contiguous range of integers. The major difficulty with providing such support is the implementation of the mixed-type operations, multiplication and division. If x and y are the integer values representing the operands, these operations reduce to finding integer approximations of αxy or $\beta x/y$ with an absolute error less than 1 for a fixed-point result type, an absolute error less than or equal to 1/2 for an integer result type, or a relative error of less than one safe interval for a floating-point result type. The static constants α and β may be arbitrary positive rational numbers, depending on the scales of the two operand types and on the result type. I present reasonably fast ways to compute appropriate fixed-point results for all values of α or β. For integer result types, I give algorithms for α and β that have the form $2^m a/b$, where a and b are positive, single-precision integer constants. Finally, I give algorithms to produce floating-point results for a variety of architectures, including the VAX, the IBM 370, and those conforming to IEEE 754.

1 Background

The Ada programming language provides support for fixed-point arithmetic. However, although the language provides for fixed-point types having arbitrary scaling factors, its standard definition only requires support of scaling factors that are powers of two, mostly out of concern for the difficulties in supporting some of the primitive operations in the general case. In this paper, I will address what are perhaps the major problems in supporting arbitrary scales: dealing with the results of fixed-point multiplications and divisions.

An Ada fixed-point type has a domain containing the set of *safe numbers*

$$\{z \cdot \sigma \mid l \leq z \leq k,\ z \text{ an integer}\}$$

where the rational quantity $\sigma > 0$ (the *small* value of the type), and the integers $l \leq k$ are constants determined by the definition of the type. A fixed-point type may contain arbitrarily many other values than these; however, the semantics puts constraints on the results of arithmetic operations in terms of the safe numbers. Given two fixed-point quantities P and Q and a fixed-point type T, Ada requires that the results of $T(P * Q)$ and $T(P/Q)$ (with $*$ and / here denoting the Ada operators) be bracketed by the greatest safe number of T not exceeding and the least safe number of T not exceeded by the mathematically correct result. Readers familiar with Ada will notice that I make no mention of the *model numbers*—a subset of the safe numbers defined by the Ada Standard. They are superfluous for most purposes, including the present ones.

In most of this paper, I will consider the *representation types* used to implement fixed-point types specified by the programmer. The defining characteristic of a representation type is that its domain comprises precisely its safe numbers. The safe numbers specified by the programmer's type and representation definitions will be integral multiples of the representation type's safe numbers.

Since we are dealing with representation types, we can consider fixed-point operands to be themselves represented by integers. If x and y are integers such that $P = x \cdot \sigma_x$, $Q = y \cdot \sigma_y$, and if the small value for the result fixed-point (representation) type T is σ_r, then the computation necessary to compute $T(P * Q)$ is to find an integer r satisfying the following inequality.

$$|r - \alpha xy| < 1, \text{ where } \alpha = \frac{\sigma_x \sigma_y}{\sigma_r}. \tag{1}$$

(As a consequence, if αxy is an integer—and thus $P * Q$ is a safe number of T—then $r = \alpha xy$.) By taking $Q = \sigma_y = 1$, this same formula indicates the criteria for converting values of one fixed-point type to another. The

computation necessary to compute $T(P/Q)$ is to find s satisfying

$$\left| s - \frac{\beta x}{y} \right| < 1, \text{ where } \beta = \frac{\sigma_x}{\sigma_y \sigma_r}. \tag{2}$$

When α and β are powers of two (which is all that an implementation is required to support), these computations are relatively straightforward. When the small values may be arbitrary rationals, however, the problem is considerably more subtle. It might at first appear, for example, that multiplication requires a computation such as axy/b where a and b are integers with $\alpha = a/b$. It would be unfortunate to be forced to compute a triple-length integer numerator (requiring three times as many bits as a, x, or y), since integer operations that produce and divide triple-length integer values usually are not supported on standard hardware directly. Likewise, division might appear to require the computation $a'x/b'y$, where $\beta = a'/b'$, as a division of a double-length integer by another double-length integer—again an operation not commonly supported. Froggatt hypothesized precisely these difficulties [1].

These difficulties, however, are not really present. Previously, J.-P. Rosen [2] has shown how to compute the desired quantities for values of α and β of the form $2^i 5^j$. In section 3, I will show that the computations of r and s do not require such unusual operations for any α or β. Instead, for fixed-point results they require an integer multiplication operation that takes two $(n+1)$-bit signed integers ($n > 0$) to produce a $(2n+2)$-bit signed product, an integer division operation that takes a $(2n+2)$-bit signed dividend and an $(n+1)$-bit signed divisor to produce an $(n+1)$-bit signed quotient, and operations for $(n+1)$-bit and $(2n+2)$-bit integer addition, subtraction, and arithmetic shifts. Section 6 contains some faster or simpler algorithms that apply when accuracy requirements are coarser (as when the implementation chooses a left-justified representation with trailing bits on the right).

Fixed-point multiplications and divisions whose results are converted to integer types pose a harder problem. This is because the semantics of Ada require a result that is correctly rounded to the nearest integer. To compute either INTEGER($P * Q$) or INTEGER(P/Q), one must find integers r' and s' respectively satisfying inequalities (3) and (4).

$$|r' - \alpha xy| \leq 1/2, \text{ where } \alpha = \sigma_x \sigma_y \tag{3}$$

$$\left| s' - \frac{\beta x}{y} \right| \leq 1/2, \text{ where } \beta = \frac{\sigma_x}{\sigma_y}. \tag{4}$$

It is *not* sufficient to perform the same computation used to satisfy (1) and (2) with $\sigma_r = 1/2$, rounding the result to INTEGER. Rounding a quantity w even to the nearest $1/2$ and then rounding that result to the nearest

integer does not necessarily yield the same result as rounding w to the nearest integer directly

Hence, (3) and (4) do require the computation of quantities such as axy/b or $a'x/b'y$. However, these computations usually can be carried out without resorting to full multiword arithmetic, as shown in section 4. In addition to the operations listed above, they require an integer remainder (provided as a result of integer division on many machines).

The expense of integer conversions may still be deemed high, even for the worst case. However, it need not be incurred for problems that do not require rounding to nearest. The programmer may always convert instead to a fixed-point type with a *small* value of 1, at the cost of rounding only to within one unit rather than half a unit. Conversion of such a type to an integer type involves no additional expense.

When the result of a fixed-point multiplication or division is to be converted to one of the floating-point types, the accuracy requirement becomes a bound on the relative error rather than the absolute error. More precisely, if FT is a floating-point type defined to have $N_d > 0$ decimal digits of accuracy, then its safe numbers are defined to comprise all numbers, h, expressible with no more than N significant binary bits in the range $2^{EMIN-1} \leq h < 2^{EMAX}$. In Ada terminology, the integers N, $EMIN$, and $EMAX$ are called, respectively, *FT'MANTISSA*, *FT'SAFE_EMIN*, and *FT'SAFE_EMAX*. The language defines $N = \lceil N_d \log_2 10 \rceil + 1$. As for fixed-point types, the domain of F may contain other numbers as well.

The value of the Ada expression $FT(P * Q)$ may be any floating-point number f satisfying the following condition.

$$f \in [\alpha xy]_N, \text{ where } \alpha = \sigma_x \sigma_y. \tag{5}$$

The notation $[\alpha xy]_N$ denotes the *N-bit safe interval containing* αxy—that is, the smallest interval containing αxy whose end-points are N-bit safe numbers. This condition implies, in particular, that f must be a safe number if αxy is. Similarly, the value of $FT(P/Q)$ may be any g satisfying

$$g \in [\beta x/y]_N, \text{ where } \beta = \sigma_x/\sigma_y. \tag{6}$$

Section 5 gives algorithms for computing f and g under a variety of assumptions about the properties of the floating-point arithmetic provided by a system.

2 Notation and Assumptions

Assume that machine arithmetic operates on signed two's-complement words of length $n + 1$ bits, so that the representable single-length integers are in

the range -2^n to $2^n - 1$. Certain operations take or produce double-length integer values in the range -2^{2n+1} to $2^{2n+1} - 1$.

In the following sections, lower-case roman letters used to denote integer variables or values will, by convention, be restricted to the range -2^n to $2^n - 1$. Upper-case roman letters that denote integers will be restricted to the range -2^{2n+1} to $2^{2n+1} - 1$. Letters f–h generally denote floating-point variables. Greek letters denote rational values.

I use an Ada-like notation for expressing algorithms. The construct

if *condition* **then** *actions* **orif** *condition* **then** $\cdots$ **end if**;

has the semantics of Dijkstra's guarded commands. The ordering of the conditions is not significant; the statement may execute the actions associated with any one of the true conditions, nondeterministically chosen at the compiler-writer's convenience. I use this notation to indicate cases where several different variations of an algorithm are applicable.

The following functions will be useful.

$$\begin{aligned}
\mathrm{sign}_+(\gamma) &= \begin{cases} 1, & \text{if } \gamma \geq 0; \\ 0, & \text{if } \gamma < 0. \end{cases} \\
\mathrm{sign}_-(\gamma) &= \mathrm{sign}_+(\gamma) - 1 = \begin{cases} 0, & \text{if } \gamma \geq 0; \\ -1, & \text{if } \gamma < 0. \end{cases} \\
\mathrm{sign}(\gamma) &= \mathrm{sign}_+(\gamma) + \mathrm{sign}_-(\gamma) \\
\mathrm{trunc}(\gamma) &= \mathrm{sign}(\gamma) \cdot \lfloor |\gamma| \rfloor \\
\mathrm{top}(\gamma) &= \mathrm{sign}(\gamma) \cdot \lceil |\gamma| \rceil
\end{aligned}$$

The standard interval notation ($[\gamma_1, \gamma_2]$, $(\gamma_1, \gamma_2]$, etc.) denotes closed, half-open, and open intervals on the real line. The notation $[\gamma \pm \delta]$ is short for $[\gamma - \delta,\ \gamma + \delta]$; likewise $[\gamma \pm \delta)$, etc.

Define $2^k\tilde{\gamma} = \gamma$ to be the *normal form* of γ if $k = \tilde{\gamma} = 0$ or $1/2 \leq |\tilde{\gamma}| < 1$. Define

$$\begin{aligned}
\mathrm{chop}(\gamma, m) &= 2^{k-m}\mathrm{trunc}(2^m\tilde{\gamma}) \\
\mathrm{top}(\gamma, m) &= 2^{k-m}\mathrm{top}(2^m\tilde{\gamma}) \\
\mathrm{round}_\lambda(\gamma, m) &= 2^{k-m}\mathrm{trunc}(2^m\tilde{\gamma} + \mathrm{sign}(\gamma)\lambda), \quad 0 \leq \lambda < 1 \\
\mathrm{round}(\gamma, m) &= \mathrm{round}_{1/2}(\gamma, m) \\
[\gamma]_m &= \begin{cases} [\mathrm{chop}(\gamma, m), \mathrm{top}(\gamma, m)], & \gamma \geq 0 \\ [\mathrm{top}(\gamma, m), \mathrm{chop}(\gamma, m)], & \gamma < 0 \end{cases}
\end{aligned}$$

That is, $\mathrm{chop}(\gamma, m)$ and $\mathrm{top}(\gamma, m)$ are the results of rounding γ to m significant bits toward 0 and toward $\pm\infty$, and $\mathrm{round}(\gamma, m)$ is the result of

rounding γ to nearest in m significant bits, with ties biased toward $\pm\infty$. The more general $\text{round}_\lambda(\gamma, m)$, spoken as "$\gamma$ rounded to m bits with bias λ," is an explicitly-biased rounding, in which (roughly speaking), numbers are rounded up to the next-higher m-bit number if their magnitude falls short of it by no more than λ units in the last place.

Borrowing from the terminology of IEEE 754, define

$$\text{logb}(\gamma) = k + 1 = \lfloor \log_2 |\gamma| \rfloor, \ \gamma \neq 0.$$

For manipulating radix 16 numbers, it will be convenient to define

$$\begin{aligned} \text{adj}^h(\gamma) &= \begin{cases} 0, & \text{if } \gamma = 0; \\ (3 - \text{logb}(\gamma)) \bmod 4, & \text{otherwise.} \end{cases} \\ \text{round}^h(\gamma, m) &= \text{round}(\gamma, m - \text{adj}^h(\gamma)) \end{aligned}$$

By analogy with binary normalization, the hexadecimal normal form of γ is $16^{k'}\tilde{\gamma}'$, where $k' = \tilde{\gamma}' = 0$ or $1/16 \leq |\tilde{\gamma}'| < 1$. For $\gamma \neq 0$, $\text{adj}^h(\gamma)$ is the number of leading 0's in $\tilde{\gamma}'$. Thus, $\text{adj}^h(1) = 3$, $\text{adj}^h(1/2) = 0$, $\text{adj}^h(64) = 1$, etc. The quantity $\text{round}^h(\gamma, m)$ is $16^{k'}$ times the result of rounding $\tilde{\gamma}'$ at the m^{th} bit after the binary point. Thus $\text{round}^h(\text{round}(\gamma, m'), m) = \text{round}(\gamma, m')$ for $0 < m' \leq m - 3$.

The machine is expected to provide, in some form, the operations listed in Figure 1 (some machines may require several instructions to implement them). These operations are undefined for operands that would produce a result outside the specified range (by the conventions above, '$\mapsto w$' indicates a single-length result and '$\mapsto W$' indicates a double-length result). In a correct implementation, such operands should arise only where the Ada semantics indicate the operation to be erroneous or the result to be undefined or exceptional.

Other operations in algorithms are denoted with standard mathematical notation. Multiplications such as $\text{sign}(xy)(a - 1)$ are not represented using '$*$', since they are merely notational devices for indicating an operation that would actually be implemented with a conditional test. Likewise, expressions such as $2^{-m-1} - \text{sign}_+(xy)$ use a standard mathematical '$-$', since the actual operation simply involves selecting one of two constants based on the signs of x and y.

3 Operations with Fixed-Point Results

Equations (1) and (2) give the general form for the problems of performing fixed-point multiplication and division with conversion to an arbitrary scale. Algorithms 1 and 2 give machine operations sufficient to carry out these computations.

- $v_1 * v_2 \mapsto W$ (integer multiplication).
- $V_1 \div v_2 \mapsto w$ (integer division, defined $w = \text{trunc}(V_1/v_2)$).
- V_1 **rem** $v_2 \mapsto w$ (remainder, defined $w = V_1 - (V_1 \div v_2) \cdot v_2$).
- $\text{shift}(V_1, m) \mapsto V_2$ (arithmetic shift, defined $V_2 = \lfloor 2^m V_1 \rfloor$).
- $v_1 \pm_s v_2 \mapsto w$, $V_1 \pm_d V_2 \mapsto W$ (single- and double-length addition or subtraction).
- $\text{floor}(v_1, m) = 2^m \lfloor 2^{-m} v_1 \rfloor$, for $m > 0$. On 2's complement machines, this is simply the result of a logical 'and' of v_1 with -2^m.
- $\text{odd}(V_1) \mapsto w$ (1 if V_1 is odd, 0 otherwise.)
- $\text{halfabs}(v_1) \mapsto w$ (half absolute value, defined $\lfloor |v_1|/2 \rfloor$). Care is needed for the special case $v_1 = -2^n$.
- $f_1 \otimes f_2 \mapsto f$ (floating-point multiplication). Subscripts will indicate specific rounding modes: $\otimes_r$ (rounded to nearest, ties resolved in any direction) or $\otimes_c$ (chopped). Without a superscript, the operators use binary normalization; the operator $\otimes_c^h$ is chopped arithmetic with hexadecimal normalization. When used without super- or subscripts, the meaning is "the standard $\otimes$ of the floating-point system being discussed." This paper considers only chopped and rounded arithmetic.
- $f_1 \oslash f_2 \mapsto f$ (floating-point division). Subscripts and superscripts have the same meaning as for $\otimes$.

Figure 1: Shorthand notations for required machine operations.

To find r satisfying $|r - \alpha xy| < 1$, given $\alpha > 0$ and $-2^n \leq \alpha xy \leq 2^n - 1$.

```
-- Assume that 2^k ξ is the normal form of round(1/α, n).
a := 2^n ξ;  m := n − k;
-- Now 2^{n−1} ≤ a < 2^n and |ε| < 2^{−n}, where α(1 + ε) = 1/2^k ξ = 2^m/a.
if m < −n − 1 then
        r := 0;
else
        v := 0;   W := 0;
        if ε < 0 then
                v := sign(xy)(a − 1);
        end if;
        if m < 0 and |ε| ≥ 2^{−n} (a−1+2^m)/a then
                W := sign_+(−εxy)(2^{−m} − 1);
                -- W = 0 is acceptable also if it is known that |xy|
                -- must be less than 2^n (a−1+2^m)/a.
        end if;
        -- Statically known whether v ≡ 0 and whether W ≡ 0.

        r := (shift(x * y +_d W, m) +_d v ) ÷ a;
end if;
```

Algorithm 1: Multiplication of fixed-point quantities, yielding a fixed-point result.

3.1 Multiplication: Algorithm 1

The basic idea behind this algorithm is to avoid having to multiply a double-precision value (xy) by turning the multiplication by α into a division. We first approximate α sufficiently closely by $2^m/a$, where a is a single-precision integer. It is always possible to do this with a relative error (denoted ϵ in the algorithms) of magnitude strictly less than 2^{-n}. Since x and y are single-precision quantities, we can compute xy exactly with the machine operation $x * y$. Shifting this result by m bits and dividing by a will now approximate the desired result, but some adjustments may be needed to get the required accuracy, depending on m and ϵ.

It is easy to verify that the algorithm works when $xy = 0$; therefore, assume $xy \neq 0$ in the following discussion. When $m < -n - 1$,

$$|\alpha xy| = \left| \frac{2^m \cdot xy}{a \cdot (1 + \epsilon)} \right| < \frac{2^{-n-2} 2^{2n}}{2^{n-1} \cdot 1/2} = 1,$$

which means that 0 is one of the end-points of the resulting safe interval,

and may be selected as the result. When $W > 0$ and $-W \le xy < 0$,

$$|\alpha xy| = \left|\frac{2^m xy}{a}\right| < \frac{2^m 2^{-m}}{2^{n-1}} \ll 1,$$

indicating that $r = 0$, which the algorithm returns in this case, is an acceptable value. Assume in the following, therefore, that $\text{sign}(xy) = \text{sign}(xy+W)$.

For the remaining cases, we must first establish that $2^m/a$ is indeed a sufficiently close approximation to α. By construction, $1/\alpha = 2^{-m}(a+\mu)$, where $|\mu| \le 1/2$ and $2^{n-1} \le a < 2^n$. Thus,

$$\alpha(1+\epsilon) = 2^m(1+\epsilon)/(a+\mu) = 2^m/a,$$

whence $\epsilon = \mu/a$. This will force $|\epsilon| < 2^{-n}$, unless $\mu = \pm 1/2$ and $a = 2^{n-1}$. If $a = 2^{n-1}$, then the definition of normalization requires $\mu > -1/2$ (this is because, as for floating-point, the normalized number next below 2^{p+n-1} is $2^{p-1}(2^n - 1)$). Since the rounding used in this treatment is biased toward infinity, furthermore, we also never achieve the case $\mu = 1/2$. Hence, $|\mu| < 1/2$ and $|\epsilon| < 2^{-n}$ in all cases.

The main assignment to r produces the result

$$r = ((xy+W)2^m - \rho_2 + v)/a - \rho_1 = (1+\epsilon)\alpha xy + .2^m W/a - \rho_2/a + v/a - \rho_1$$

where $\text{sign}(xy)\rho_1 \in [0, 1 - 1/a]$ and $\rho_2 \in [0, 1 - 2^m]$, as given by Facts 1 and 2 in the Appendix. In other words, the absolute error in r is

$$\delta = r - \alpha xy = \epsilon\alpha xy - \rho_2/a - \rho_1 + (2^m W + v)/a$$

For brevity, define $\eta = (2^m W + v)/a$ and $\hat{\epsilon} = 2^n|\epsilon| < 1$. Because $|\alpha xy| \le 2^n$, assume that $|\epsilon\alpha xy| \le \hat{\epsilon}$. This allows us to bound δ as follows.

	xy	m	ϵ	δ
1.	> 0	≥ 0	≥ 0	$[-1 + 1/a + \eta,\ \hat{\epsilon} + \eta]$
2.	< 0	≥ 0	≥ 0	$[-\hat{\epsilon} + \eta,\ 1 - 1/a + \eta]$
3.	> 0	< 0	≥ 0	$[-1 + 2^m/a + \eta,\ \hat{\epsilon} + \eta]$
4.	< 0	< 0	≥ 0	$[-\hat{\epsilon} - 1/a + 2^m/a + \eta,\ 1 - 1/a + \eta]$
5.	> 0	≥ 0	< 0	$[-\hat{\epsilon} - 1 + 1/a + \eta,\ \eta)$
6.	< 0	≥ 0	< 0	$(\eta,\ \hat{\epsilon} + 1 - 1/a + \eta]$
7.	> 0	< 0	< 0	$[-\hat{\epsilon} - 1 + 2^m/a + \eta,\ \eta)$
8.	< 0	< 0	< 0	$(-1/a + 2^m/a + \eta,\ \hat{\epsilon} + 1 - 1/a + \eta]$

The width of each of these intervals for δ is strictly less than 2. The idea behind Algorithm 1 is to choose W and v, and thus η, so as to keep this interval within $(-1, 1)$, which guarantees the desired post-condition.

By inspection, cases 1, 2, and 3 always yield the correct interval with $\eta = 0$. Case 4 yields the correct interval if $\hat{\epsilon} < 1 - 1/a + 2^m/a = (a - 1 + 2^m)/a$

To find s satisfying $|s - \beta x/y| < 1$, given $\beta > 0$ and $-2^n \leq \beta x/y \leq 2^n - 1$.

```
-- Assume that 2^k ξ is the normal form of round(β, n).
b := 2^n ξ;  m := k − n;
-- Now 0 < b < 2^n and |ε| ≤ 2^−n, where β(1 + ε) = 2^m b.
if  m ≤ −2n or x = 0 then
        s := 0;
else
        W := 0;  v := 0;
        if m < 0 then
                W := sign+(−x) (2^−m − 1);
        end if;
        if ε < 0 then
                v := sign(xy);
        end if;
        -- Statically known whether v ≡ 0 and whether W ≡ 0.

        s := v +s shift(b * x +d W, m) ÷ y;
end if
```

Algorithm 2: Division of fixed-point quantities, yielding a fixed-point result.

and $\eta = 0$. In each of these cases, Algorithm 1 sets v and W (and thus η) to 0. Cases 5, 6, and 8 will work if we use η to move the error interval toward infinity (i.e., in the direction of the sign of xy) by an amount of magnitude $1 - 1/a$. Again, this same use of η will also work for case 7 if $\hat{\epsilon} < 1 - 1/a + 2^m/a$. Thus in each of these cases, Algorithm 1 sets v to $\text{sign}(xy)(a - 1)$ and W to 0.

This leaves cases 4 and 7 when $1 - 1/a + 2^m/a \leq \hat{\epsilon} < 1$. In case 4, setting η to $1/a - 2^m/a = (1 - 2^m)/a$ will move the interval sufficiently far. This is accomplished by setting $v = 0$ and $W = 2^{-m} - 1$. In case 7, it suffices to set η to $1 - 2^m/a = 1 - 1/a + (1 - 2^m)/a$, which is accomplished by setting $v = a - 1$ and $W = 2^{-m} - 1$.

3.2 Division: Algorithm 2

For division, the computation is actually a bit more straightforward than for multiplication; we actually multiply x by an approximation of β (separated into a multiply and a shift) and then divide by y with the adjustments needed to roughly center the error interval around 0. The suitability of $2^m b$ as an approximation is immediate in this case (Fact 7)).

The case $x = 0$ is trivial, so assume $x \neq 0$ in the following. When

$m \le -2n$,

$$|\beta x/y| = |2^m \cdot b \cdot (1+\epsilon) \cdot x/y| \le 2^{-2n} \cdot (2^n - 1) \cdot (1 + 2^{-n}) \cdot 2^n < 1,$$

allowing the choice $s = 0$. If $m < 0$ and $-2^{-m} + 1 \le bx < 0$, then

$$0 > \frac{\beta x}{|y|} = \frac{2^m bx}{|y|(1+\epsilon)} \ge \frac{-1 + 2^m}{|y|(1+\epsilon)} > \frac{-1 + 2^m}{1+\epsilon}.$$

Thus, $\beta x/|y| \in (-2, 0)$ if $\epsilon < 0$, allowing the approximation $s = \text{sign}(xy)$ provided by the algorithm in that case. Likewise, $\beta x/|y| \in (-1, 0)$ if $\epsilon \ge 0$, which allows 0 as a result. In the rest of the discussion, accordingly, assume $\text{sign}(x) = \text{sign}(bx) = \text{sign}(bx + W)$.

The main computation of s produces the result

$$s = v + (2^m(bx + W) - \rho_2)/y - \rho_1 = \beta(1+\epsilon)x/y + v + 2^m W/y - \rho_2/y - \rho_1,$$

where again ρ_1 and ρ_2 are as given by Facts 1 and 2. The error term is therefore

$$\delta = s - \beta x/y = \epsilon\beta x/y + v + 2^m W/y - \rho_2/y - \rho_1.$$

Define $\eta(y) = v + 2^m W/y$ and again take $\hat{\epsilon} = 2^n|\epsilon|$, so that $|\epsilon\beta x/y| \le \hat{\epsilon}$.

When $m \ge 0$ ($\rho_2 = 0$), the following bounds hold for δ.

	xy	ϵ	δ
1.	> 0	≥ 0	$[-1 + 1/\lvert y\rvert + \eta(y),\quad \hat{\epsilon} + \eta(y)]$
2.	< 0	≥ 0	$[-\hat{\epsilon} + \eta(y),\quad 1 - 1/\lvert y\rvert + \eta(y)]$
3.	> 0	< 0	$[-\hat{\epsilon} - 1 + 1/\lvert y\rvert + \eta(y),\quad \eta(y))$
4.	< 0	< 0	$(\eta(y),\quad \hat{\epsilon} + 1 - 1/\lvert y\rvert + \eta(y)]$

The widths of all these intervals are strictly less than 2 for all valid values of y ($1 \le |y| \le 2^n$). In cases 1 and 2, no further adjustment is needed, and the algorithm sets $\eta(y) = v = W = 0$. Cases 3 and 4 require that the interval be moved by 1 in the direction of the sign of the result, which is accomplished by setting $W = 0$ and $\eta(y) = v = \text{sign}(xy)$. We can get away with adjusting the interval by precisely 1 (rather than slightly less) because of the strict inequality $\epsilon < 0$ in these cases, which makes the error interval open on one side.

Consider now $m < 0$, for which we have the following bounds on δ.

	xy	x	ϵ	δ
5.	> 0	> 0	≥ 0	$[-1 + 2^m/\lvert y\rvert + \eta(y),\quad \hat{\epsilon} + \eta(y)]$
6.	> 0	< 0	≥ 0	$[-1 + 1/\lvert y\rvert + \eta(y),\quad \hat{\epsilon} + 1/\lvert y\rvert - 2^m/\lvert y\rvert + \eta(y)]$
7.	< 0	> 0	≥ 0	$[-\hat{\epsilon} + \eta(y),\quad 1 - 2^m/\lvert y\rvert + \eta(y)]$
8.	< 0	< 0	≥ 0	$[-\hat{\epsilon} - 1/\lvert y\rvert + 2^m/\lvert y\rvert + \eta(y),\quad 1 - 1/\lvert y\rvert + \eta(y)]$
9.	> 0	> 0	< 0	$[-\hat{\epsilon} - 1 + 2^m/\lvert y\rvert + \eta(y),\quad \eta(y))$
10.	> 0	< 0	< 0	$[-\hat{\epsilon} - 1 + 1/\lvert y\rvert + \eta(y),\quad 1/\lvert y\rvert - 2^m/\lvert y\rvert + \eta(y))$
11.	< 0	> 0	< 0	$(\eta(y),\quad \hat{\epsilon} + 1 - 2^m/\lvert y\rvert + \eta(y)]$
12.	< 0	< 0	< 0	$(-1/\lvert y\rvert + 2^m/\lvert y\rvert + \eta(y),\quad \hat{\epsilon} + 1 - 1/\lvert y\rvert + \eta(y)]$

The intervals in cases 5 and 7 require no adjustment; we can take $\eta(y) = W = v = 0$. In cases 6 and 8, the interval extends too far in the direction of the sign of $-y$ by as much as $(1 - 2^m)/|y|$; setting W to $2^{-m} - 1$ in these cases gives $\eta(y) = (1 - 2^m)/y = \text{sign}(y)(1 - 2^m)/|y|$. In cases 9 and 11, the interval may be corrected by setting $W = 0$ and $\eta(y) = v = \text{sign}(xy)$. In cases 10 and 12 the intervals can extend too far by $1 - 1/|y|$ in the direction of $-\text{sign}(xy) = \text{sign}(y)$. The algorithm sets $v = \text{sign}(xy)$ and $W = 2^{-m} - 1$, which give the error intervals $[-\hat{\epsilon} + 2^m/|y|, 1)$ for case 10 and $(-1, \hat{\epsilon} - 2^m/|y|]$ for case 12. Setting $W = 2^{-m}$ (equivalently, subtracting 1 from $\text{shift}(b * x, m)$) would also have worked for this case.

3.3 Implementation notes on Algorithms 1 and 2

- The quantities a, b, m, and ϵ may all be computed at compilation. The computations of W and v can be carried out, where needed, knowing only the signs of x and y.
- The Ada Standard only requires handling the cases where $\epsilon = 0$ and $a = 1$ ($b = 1$), for which division (multiplication) other than shifting is unnecessary.
- If αxy or $\beta x/y$ is out of range, Algorithms 1 and 2 may either result in an overflow or a (mathematically) incorrect result. The semantics of Ada don't specify any particular behavior in these cases either.

4 Operations with Integer Results

Equations (3) and (4) give the general form for the problems of obtaining correctly rounded integer results of fixed-point multiplication and division. Algorithm 3 and Algorithm 4 find satisfactory values for r' and s'. Unfortunately, they do not accept arbitrary positive α or β for these algorithms. Rather, α and β must be expressible in the form $2^m a/b$ with $0 < a, b < 2^n$.

Although these algorithms look more formidable than those for fixed-point results—and are certainly more expensive—they are less subtle. In essence, both of them compute their respective quotients ($2^m axy/b$ and $2^m ax/by$) after adding in "fudge factors" to get proper rounding. Where necessary they perform, in effect, a certain amount of multiple-precision arithmetic by breaking the constant into high- and low-order parts.

4.1 Multiplication: Algorithm 3

When $\alpha \leq 2^{-2n-1}$, we have $|xy| \leq 1/2$, to which 0 is always a valid approximation. When $m \geq 0$ and $2^m a \geq b$, we know that xy must be single-precision unless the final result will overflow. Therefore, it is safe to perform

To find r' satisfying $|r - \alpha xy| \leq 1/2$, given $\alpha = 2^m a/b$; $0 < a, b \leq 2^n - 1$; and $-2^n \leq \alpha xy \leq 2^n - 1$. Assume without loss of generality that either $m \geq 0$ and $2^m a \geq b$ or that $m \leq 0$ and $a < 2b$.

if $\alpha \leq 2^{-2n-1}$ **then**
    $r' := 0$;
elsif $m \geq 0$ **and** $2^m a \geq b$ **then**
    $V := \text{shift}((x * y) * a,\ m)$;
    $r' := (\ V +_{\text{d}} \text{sign}(xy)\lfloor b/2 \rfloor\ ) \div b$;
elsif $m = 0$ **and** $a < b$ **then**
    $v := (a * x \div b) * y$; $W := (a * x \textbf{ rem } b) * y$;
    $r' := v +_{\text{s}} (\ W +_{\text{d}} \text{sign}(xy)\ \lfloor b/2 \rfloor\ ) \div b$;
elsif $m < 0$ **and** $a < b$ **then**
    $V := (a * x \div b) * y +_{\text{d}} (a * x \textbf{ rem } b) * y \div b$;
    $r' := \text{shift}(V +_{\text{d}} (2^{-m-1} + \text{sign\_}(xy)),\ m)$;
elsif $m < 0$ **and** $b \leq a < 2b$ **then**
    $V := x * y +_{\text{d}} ((a-b) * x \div b) * y$
        $+_{\text{d}} ((a-b) * x \textbf{ rem } b\ ) * y \div b$;
    $r' := \text{shift}(V +_{\text{d}} (2^{-m-1} + \text{sign\_}(xy)),\ m)$;
end if;

Algorithm 3: Multiplication of fixed-point quantities, yielding an integer result.

the obvious computation. Only the division by b introduces error; the addition of $\text{sign}(xy)\lfloor b/2 \rfloor$ before the division has the effect of rounding the result to nearest (see Fact 3) in the Appendix). For $m \geq 0$, the only remaining case is $m = 0$ and $2^m a = a < b$. Here, the algorithm uses the identity $axy/b = (ax \div b)y + (ax \textbf{ rem } b)y/b$, rounding the computation of the second term. The divisions cannot overflow, since $|ax/b| < |x|$ and $|(ax \textbf{ rem } b)y/b| < |y|$.

Consider now $m < 0$. If $a < b$, the computation of ax/b must yield a single-precision result, and the algorithm again uses the identity $axy/b = (ax \div b)y + (ax \textbf{ rem } b)y/b$, getting a correctly rounded result out of the subsequent right shift by using Fact 4) from the Appendix. The same strategy works when $b \leq a < 2b$, but to prevent overflow, the computation of V uses $axy/b = xy + (a - b)xy/b$, where the computation of the second term works as for the case $a < b$ (since $a - b < b$).

4.2 Division: Algorithm 4

If $\beta \leq 2^{-n-1}$, then $\beta x/y \leq 1/2$, which allows the approximation $s' = 0$. If $\beta \geq 2^{2n}$, all non-zero results are at or outside $[-2^n, 2^n - 1]$, and we can

To find s' satisfying $|s' - \beta x/y| \leq 1/2$, given $\beta = 2^m a/b$, $a, b > 0$, and $-2^n \leq \beta x/y \leq 2^n - 1$. Assume without loss of generality that either $m \geq 0$ or $a < 2b$.

if $\beta \leq 2^{-n-1}$ **or** $x = 0$ **then**
    $s' := 0;$
elsif $\beta \geq 2^{2n}$ **then**
    $s' := \text{sign}(xy)(2^n - 1) +_s \text{sign\_}(xy);$
elsif $-n \leq m < 0$ **and** $a = b$ **and** $y = -1$ **then**
    $s' := \text{shift}((2^{-m-1} + \text{sign\_}(xy)) -_d x,\ m);$
elsif $-n \leq m < 0$ **and** $a \leq b$ **then**
    $s' := \text{shift}((a * x \div b) \div y +_d (2^{-m-1} + \text{sign\_}(xy)),\ m);$
elsif $-n - 1 \leq m < 0$ **and** $b < a < 2b$ **then**
    $T := x +_d (a-b) * x \div b;$
    if $y = \pm 1$ **then**
        $s' := \text{shift}(\text{sign}(y)T +_d (2^{-m-1} + \text{sign\_}(xy)),\ m);$
    else
        $s' := \text{shift}(T \div y +_d (2^{-m-1} + \text{sign\_}(xy)),\ m);$
    end if;
elsif $m \geq 0$ **then**
    $d = 2^m a \mod b;$
    $v := 0;$
    if $\text{odd}(y)$ **then**
        $v := \text{sign}(x)\lfloor (b-1)/2 \rfloor;$ $--$ 0 if $b \leq 2$.
    end if;
    $u := (d * x +_d v) \div b;$
    $U := \text{shift}(\lfloor \beta/2^n \rfloor * x, n) +_d (\lfloor \beta \rfloor \bmod 2^n) * x;$
    $--$ U reduces to $\lfloor \beta \rfloor * x$ when $\beta < 2^n$.
    $s' := (U +_d u +_d \text{sign}(x)\text{halfabs}(y)) \div y;$
end if;

Algorithm 4: Division of fixed-point quantities, yielding an integer result.

therefore arbitrarily return the extreme values of s' for non-zero x. By assumption, the remaining cases have $m \geq -n-1$. We need consider only $x \neq 0$, since $x = 0$ is obviously correct. For convenience, this algorithm gives $0/0 = 0$, which Ada allows, but which an implementation might want to handle differently.

Consider first $m < 0$. If $a < b$, then ax/b must be a single-precision quantity, and we can simply divide by y and shift, after adding a correction of $2^{-m-1} + \text{sign\_}(xy)$ to cause rounding (Facts 4 and 5). When $a = b$ the same computation works, but must be re-organized in the case $y = -1$ to avoid overflow when $x = -2^n$. When $b < a < 2b$, the algorithm uses the identity $xa/b = x + x(a-b)/b$, adds the usual correction term, and shifts. As long as $|y| \geq 2$, the computation of $\text{trunc}(ax/by)$ will produce a single-precision result; the only possible problem occurs when $|y| = 1$, so that is treated as a special case.

Now assume $m \geq 0$. The algorithm uses the identities

$$2^m a/b = \beta = \lfloor\beta\rfloor + (2^m a \bmod b)/b$$
$$\lfloor\beta\rfloor = 2^n\lfloor\beta/2^n\rfloor + \lfloor\beta\rfloor \bmod 2^n$$

to break the computation of $2^m ax/b$ into sufficiently small pieces. The resulting error in s' is given by

$$\begin{aligned}\delta &= s' - \beta x/y = \text{sign}(xy)(-\rho/|y| - \rho' + \lfloor|y|/2\rfloor/|y| + |v/by|), \\ &\quad \text{where } 0 \leq \rho \leq 1 - 1/b,\ 0 \leq \rho' \leq 1 - 1/|y|.\end{aligned}$$

This gives

$$\begin{aligned}\text{sign}(xy)\delta &\geq -1/|y| + 1/b|y| - 1 + 1/|y| + \lfloor|y|/2\rfloor/|y| + |v/by| \\ &= -1 + 1/b|y| + \lfloor|y|/2\rfloor/|y| + |v/by| \\ \text{sign}(xy)\delta &\leq \lfloor|y|/2\rfloor/|y| + |v/by|\end{aligned}$$

When y is even, the algorithm sets $v = 0$ and $\lfloor|y|/2\rfloor = |y|/2$, so that

$$-1/2 + 1/b|y| \leq \text{sign}(xy)\delta \leq 1/2$$

and $|\delta| \leq 1/2$. When y is odd, the algorithm sets

$$v = \text{sign}(x)\lfloor(b-1)/2\rfloor = \text{sign}(x)(b-l)/2, \text{ for } l = 1 \text{ or } 2;$$

and in this case, $\lfloor|y|/2\rfloor = |y|/2 - 1/2$. As a result,

$$-1/2 \leq \text{sign}(xy)\delta \leq 1/2 - |1/2by|$$

and again $|\delta| \leq 1/2$.

5 Operations with Floating-Point Results

The computations necessary to produce correct floating-point depend on several parameters of the arithmetic:

- $N > 0$, the number of significand bits in the safe numbers;
- $N' > 4$, the number of significand bits available for computation;
- $n > 0$, the number of bits in the magnitude of an integer;
- the floating-point radix; and
- the available rounding modes of the floating-point multiplication and division operators.

I will present algorithms for any of the following situations.

- $N' \geq N + 3$, binary radix, with all results of multiplications and divisions rounded to nearest (ties resolved in either direction).
- $N' \geq N + 3$, radix 16, with results chopped.
- $N' \geq N+2$, binary radix, with operations to produce either all chopped results or both rounded and chopped results.

If γ is the mathematically-correct result, the strategy employed in all cases is to first compute a floating-point result $\hat{h}$ satisfying

$$\begin{aligned} \hat{h} \;=\; & \mathrm{chop}((1+\delta)\gamma, M'); \\ & N \leq M < M' \leq N', \\ & -\lambda 2^{-M} < \delta < \lambda' 2^{-M}, \text{ with} \\ & \lambda + \lambda' \leq 1,\ \lambda, \lambda' \geq 0, \text{ and } \lambda = l \cdot 2^{M-M'}, \text{ integer } l \end{aligned}$$

and then to invoke the following lemma.

Lemma 1 $\mathrm{round}_\lambda(\hat{h}, M) \in [\gamma]_M$.

In other words, the result of the biased rounding of $\hat{h}$ is a correct approximation to γ for a type with M-bit safe numbers, and therefore also for a type with N-bit safe numbers (Fact 6). The conditions on λ also serve to make the biased rounding implementable.

One very important, and possibly surprising, consequence of Lemma 1 is that if the last operation in the computation of $\hat{h}$ chops its result, then it does not contribute to δ. It is preferable, in fact, to have the last operation chop rather than round—even though the latter produces a more accurate result—if the outcome is then to be rounded to a lower precision. On the

other hand, it is generally better to have other operations produce rounded results, decreasing relative error. Thus, it is indeed useful that arithmetic conforming to IEEE 754 provides for both chopped and rounded arithmetic.

Proof of Lemma 1. The case $\gamma = 0$ is immediate; assume $\gamma \neq 0$. Define $\gamma' = \gamma(1+\delta)$ and without loss of generality, take $1/2 \leq \gamma' < 1$ (the formula is symmetric with respect to sign, and extending to magnitudes outside $[1/2, 1)$ is merely a matter of scaling). It suffices to show that $\text{round}(\gamma', M) \in [\gamma]_M$, since because of the form of λ,

$$\begin{aligned} \text{round}_\lambda(\gamma', M) &= \text{chop}(\gamma' + l \cdot 2^{-M'}, M) \\ &= \text{chop}(\text{chop}(\gamma', M') + l \cdot 2^{-M'},\ M) \\ &= \text{round}_\lambda(\text{chop}(\gamma', M'),\ M). \end{aligned}$$

Define $\eta = \gamma' + \lambda 2^{-M}$. If $\gamma \leq 1$, then

$$\begin{aligned} \gamma(1 - \lambda 2^{-M}) + \lambda 2^{-M} < \eta < \gamma(1 + \lambda 2^{-M}) + \lambda' 2^{-M} \\ \gamma < \eta < \gamma + (\lambda + \lambda') 2^{-M} \leq \gamma + 2^{-M}. \end{aligned}$$

Thus, $\text{chop}(\gamma, M) \leq \text{chop}(\eta, M)$ and, since safe numbers above η are at least 2^{-M} apart, $\text{chop}(\eta, M)$ must be no larger than the next safe number at or above γ. Thus $\text{chop}(\eta, M) \in [\gamma]_M$. This leaves the case $\gamma > 1$, which is possible only if $\delta < 0$, $\gamma < 1 + 2^{-M+1}$ and $\gamma' > 1 - \lambda 2^{-M}$. Thus, $\text{chop}(\gamma, M) = 1$ and $\text{chop}(\eta, M) = 1$, so that again $\text{chop}(\eta, M) \in [\gamma]_M$. □

The value $\text{round}(\gamma, m) = \text{round}_{1/2}(\gamma, m)$ is biased toward $\pm\infty$ in cases of ties. However, for the algorithms to follow, it will not matter whether the operations $\otimes$ and $\oslash$, when they round, use this biased rounding (as does the VAX) or unbiased rounding (rounding to even, as does IEEE 754).

In general, δ will be a product of terms having the form $(1 + \delta_i)$. When an operation involves rounding the mathematically correct result, the corresponding δ_i will satisfy $|\delta_i| \leq \mu_0$, where we define

$$\mu_j = 1/(1 + 2^{N'-j}). \tag{7}$$

When the operation involves chopping, $-\mu_1 < \delta \leq 0$.

5.1 Multiplication: Algorithm 5

Consider first the case where $\otimes_c$ is available (binary radix, chopped). Let

$$\begin{aligned} \hat{f} &= \text{round}(\alpha, N') \otimes_c \text{round}(x * y, N') = \text{chop}(\alpha x y(1 + \delta),\ N'), \\ \delta &= (1 + \delta_\alpha)(1 + \delta_{xy}) - 1 \end{aligned}$$

where $|\delta_\alpha|$ and $|\delta_{xy}|$ are bounded by μ_0. The value $\text{round}(\alpha, N')$ is a compile-time quantity. By the properties of rounding,

$$-2^{-N'+1} < (1-\mu_0)^2 - 1 \leq \delta \leq (1+\mu_0)^2 - 1 < 2^{-N'+1}.$$

As a result, rounding $\hat{f}$ to $N'-2$ bits yields a correct result, as long as $N \leq N'-2$.

For the case that only $\otimes_r$ is available, consider

$$\begin{aligned} \hat{f} &= \text{round}(\alpha, N') \otimes_r \text{round}(x * y, N') \\ &= \alpha x y (1+\delta_\alpha)(1+\delta_{xy})(1+\delta_\otimes) \\ &= \alpha x y (1+\delta), \end{aligned}$$

where the magnitudes of the subscripted δ's are again bounded by μ_0. Thus,

$$-3 \cdot 2^{-N'} < (1-\mu_0)^3 - 1 \leq \delta \leq (1+\mu_0)^3 - 1 < 3 \cdot 2^{-N'}.$$

(the outer inequalities here—and similar ones later in this paper—may be derived by fairly simple algebraic manipulation, although I confess to having used Macsyma). Since $|\delta|$ is bounded by $4 \cdot 2^{-N'}$, the value $\text{round}(\hat{f}, N'-3)$ is a valid result.

If only $\otimes_c^h$ is available (hexadecimal radix, chopped), we can use the same analysis as for $\otimes_c$, substituting μ_2 for μ_0 (that is, as if rounding to $N'-3$ bits). Thus, doing all rounding and arithmetic in hexadecimal,

$$\hat{f} = \text{round}^h(\alpha, N') \otimes_c^h \text{round}^h(x * y, N'),$$

gives a result that can be validly rounded to $N'-5$ bits. When this is insufficient precision, a simple trick will produce a correct result with $N'-3$ bits of significance (the maximum possible value of N for hexadecimal arithmetic according to Ada rules). For any real number γ,

$$\text{round}(2^m\gamma, N') = \text{round}^h(2^m\gamma, N'), \text{ where } m = \text{adj}^h(\gamma).$$

That is, hexadecimal rounding of $2^m\gamma$ yields a significand that is also normalized for binary radix. Furthermore, when two such numbers are multiplied, the hexadecimally-normalized result can have at most one leading binary 0; the result is chopped to either N' or $N'-1$ significant bits. By the previous analysis, therefore, the result can be correctly rounded to $N'-2$ bits in binary, and thus to $N'-3$ bits hexadecimal (with scaling by a power of two to correct the effects of the adjustments for binary normalization).

Algorithm 5 summarizes the analysis of fixed-point multiplications producing floating-point results.

To find a floating-point number $f \in [\alpha xy]_N$, *given* $|\alpha xy| = 0$ *or* $2^{EMIN-1} \leq |\alpha xy| \leq 2^{EMAX}(1 - 2^{-N})$. This algorithm works for (1) $N' \geq N + 3$ with $\otimes_c$, $\otimes_r$ or $\otimes_c^h$ available; or (2) $N' \geq N + 2$, with $\otimes_c$ available.

$U := x * y;$
-- Where multiple **orif** branches apply, choose any convenient one.
if $N' \geq N + 2$ **then** -- binary radix, chopped.
    $f :=$ round(round$(\alpha, N') \otimes_c$ round$(U, N'),\ N' - 2);$
orif $N' \geq N + 3$ **then** -- binary radix, rounded.
    $f :=$ round(round$(\alpha, N') \otimes_r$ round$(U, N'),\ N' - 3);$
orif $N' \geq N + 3$ **then** -- hexadecimal radix, chopped.
    $m_1 :=$ adj$^h(\alpha)$; $m_2 :=$ adj$^h(U)$;
    $\hat{f} := 2^{m_1}$round$(\alpha, N') \otimes_c^h 2^{m_2}$round$(U, N')$;
    $f := 2^{-m_1-m_2} \cdot$ round$(\hat{f}, N' - 3)$;
orif $N' \geq N + 5$ **then** -- hexadecimal radix, chopped.
    $f :=$ round(round$^h(\alpha, N') \otimes_c^h$ round$^h(U, N'),\ N' - 5);$
end if;

Algorithm 5: Multiplication of fixed-point quantities, yielding a floating-point result.

5.2 Division: Algorithms 6 and 7

The approaches used for division depend on whether integers are exactly representable ($n \leq N'$). First, let us consider the cases in which they are and in which the radix is binary. If both $\otimes_r$ and $\otimes_c$ are available (as under IEEE 754), then there is the computation

$$\hat{g} = (\text{round}(\beta, N') \otimes_r x) \oslash_c y = \text{chop}((\beta x/y)(1 + \delta_\beta)(1 + \delta_\otimes),\ N')$$

which, by the same analysis as in the preceding section, will give a valid result when rounded to $N' - 2$ bits. Replacing $\oslash_c$ with $\oslash_r$ yields three rounding errors, which may, as in section 5.1, be rounded to $N' - 3$ bits. Using chopping for all arithmetic gives

$$\begin{aligned} \hat{g} &= (\text{round}(\beta, N') \otimes_c x) \otimes_c y \\ &= \text{chop}((\beta x/y)(1 + \delta_\beta)(1 + \delta'_\otimes),\ N') \end{aligned}$$

where $\delta'_\otimes$ indicates the relative error of the chopped multiplication: $-\mu_1 < \delta'_\otimes \leq 0$. Thus, the relative error, δ, in the argument to chop is bounded by

$$-3 \cdot 2^{-N} < (1 - \mu 0)(1 - \mu 1) - 1 < \delta < \mu 0 < 2^{-N}.$$

so that rounding to $N' - 2$ bits with a bias of 3/4 gives a correct result.

To find a binary-radix floating-point number $g \in [\beta x/y]_N$, *given* $\beta x/y = 0$ *or* $2^{EMIN-1} \le |\beta x/y| \le 2^{EMAX}(1-2^{-N})$. The computation requires either (1) $N' \ge N+2$ with $\oslash_c$ and either $\otimes_c$ or $\otimes_r$ available; (2) $N' \ge N+2$ with $N' < n$; or (3) $N' \ge N+3$ with $\otimes_r$ and $\oslash_r$ available.

```
-- Where multiple orif branches apply, choose any convenient one.
if  N' ≥ N + 4 then   -- rounded or chopped.
        ĝ := (round(β, N') ⊗ round(x, N')) ⊘ round(y, N');
        g := round(ĝ, N' − 4);
orif N' ≥ N + 3 and N' ≥ n then   -- rounded.
        g := round((round(β, N') ⊗r x) ⊘r y,  N' − 3);
orif N' ≥ N + 2 and N' ≥ n then   -- chopped.
        g := round3/4((round(β, N') ⊗c x) ⊘c y,  N' − 2);
orif N' ≥ N + 2 and N' ≥ n then   -- rounded and chopped.
        g := round((round(β, N') ⊗r x) ⊘c y,  N' − 2);
orif N' ≥ N + 2 and N' < n then
        if x = 0 then
                g := 0.0;
        else
                m1 := n − logb(round(β, n − 1)) − 2;
                m2 := min(0,  n − logb(x) − 1);
                m3 := min(0,  n − logb(y) − 1);
                u := (2^m1 round(β, n − 1) * 2^m2 x) ÷ 2^m3 y;
                g := 2^(−m1−m2+m3) round(u, N' − 2);
        end if;
end if;
```

Algorithm 6: Division of fixed-point quantities, yielding a floating-point result (binary radix case).

To find a hexadecimal-radix floating-point number $g \in [\beta x/y]_N$, *using chopped arithmetic, given* $\beta x/y = 0$ *or* $2^{EMIN-1} \le |\beta x/y| \le 2^{EMAX}(1 - 2^{-N})$. The computation requires that $\otimes_c^h$ and $\oslash_c^h$ be available and that $N' \ge N + 3$.

```
-- Where multiple orif branches apply, choose any convenient one.
if  N' ≥ N + 7 then
        ĝ := (round^h(β, N') ⊗_c^h round^h(x, N')) ⊘_c^h round^h(y, N');
        g := round(ĝ, N' − 7);
orif N' ≥ N + 5 and N' ≥ n + 3 then
        ĝ := (round^h(β, N') ⊗_c^h x) ⊘_c^h y;
        g := round_{3/4}(ĝ, N' − 5);
orif N' ≥ N + 4 and N' ≥ n then
        m1 := adj^h(β);  m2 := adj^h(x);  m3 := adj^h(y);
        ĝ := (2^m1 round(β, N') ⊗_c^h 2^m2 x) ⊘_c^h 2^m3 y;
        g := 2^(−m1−m2+m3) round(ĝ, N' − 4);
orif N' ≥ N + 3 and N' ≥ n then
        m1 := adj^h(β);  m2 := adj^h(x);  m3 := adj^h(y);
        h := 2^m1 round(β, N') ⊗_c^h 2^m2 x;
        m4 := adj^h(h) − 1;   -- m3 = 0 or −1.
        ĝ := 2^m4 chop(h, N' − 1) ⊘_c^h 2^m3 y;
        g := 2^(−m1−m2+m3−m4) round_{3/4}(ĝ, N' − 3);
orif N' ≥ N + 3 and N' < n then
        if x = 0 then
                g := 0.0;
        else
                m1 := n − logb(round(β, n − 1)) − 2;
                m2 := min(0,  n − logb(x) − 1);
                m3 := min(0,  n − logb(y) − 1);
                u := (2^m1 round(β, n − 1) * 2^m2 x) ÷ 2^m3 y;
                g := 2^(−m1−m2+m3) round(u, N' − 3);
        end if;
end if;
```

Algorithm 7: Division of fixed-point quantities, yielding a floating-point result (hexadecimal radix case).

Next, consider the cases using binary radix in which $N' < n$, so that integers cannot (always) be represented exactly. If all arithmetic rounds,

$$(\text{round}(\beta, N') \otimes_r \text{round}(x, N')) \oslash_r \text{round}(y, N') = (\beta x/y)(1+\delta),$$
$$\delta = (1+\delta_\beta)(1+\delta_x)(1+\delta_\otimes)(1+\delta_\oslash)/(1+\delta_y) - 1$$

giving

$$-5 \cdot 2^{-N'} < (1-\mu_0)^4/(1+\mu_0) - 1 \le \delta \le (1+\mu_0)^4/(1-\mu_0) - 1 < 6 \cdot 2^{-N'}.$$

Thus, $|\delta| < 2^{-N'+3}$, which allows correct rounding to $N'-4$ bits. When the rounding operations are replaced by chopping,

$$\begin{aligned} & (\text{round}(\beta, N') \otimes_c \text{round}(x, N')) \oslash_c \text{round}(y, N') \\ = \ & \text{chop}((\beta x/y)(1+\delta),\ N'), \\ \delta \ = \ & (1+\delta_\beta)(1+\delta_x)(1+\delta'_\otimes)/(1+\delta_y) - 1 \end{aligned}$$

Here,

$$-5\cdot 2^{-N'} < (1-\mu_0)^2(1-\mu_1)/(1+\mu_0) - 1 \le \delta \le (1+\mu_0)^2/(1-\mu_0) - 1 < 4\cdot 2^{-N'}$$

so that again $|\delta| < 2^{-N'+3}$, and rounding to $N'-4$ bits works.

If $N' < n$ and $N'-4$ bits are too few, it is possible to perform the computation using integer operations. This first involves expressing β as an integer in the range $[2^{n-2},\ 2^{n-1}-1]$ times a power of two, similar to what was done in Algorithm 2, and likewise scaling x and y so that their magnitudes are in the range $[2^{n-1},\ 2^n]$. This gives a value of β rounded to $n-1 \ge N'$ binary digits. Performing integer multiplication by the scaled x followed by (truncated) integer division by the scaled y produces the result

$$2^m \text{chop}(\text{round}(\beta, n-1)x/y, n'), \text{ where } n-2 \le n' \le n,$$

where m reflects all the scaling. Since the only error before the final chop is from the rounding of β to at least N' bits, it is easily seen that this result may be safely rounded to any number of bits less than $N'-1 \le n-2$. Since only integer operations are involved, this same strategy will work for hexadecimal radix, if the final rounding is to $N'-3$ bits.

Now assume a hexadecimal radix. The analysis above (for binary radix) works if $N'-3$ is substituted for N'. Thus,

$$(\text{round}^h(\beta, N') \otimes_c^h \text{round}^h(x, N')) \oslash_c^h \text{round}^h(y, N')$$

gives a valid result when rounded to $N'-7$ bits, and if $N' \ge n+3$ (integers exactly representable), then

$$(\text{round}^h(\beta, N') \otimes_c^h x) \oslash_c^h y$$

will give a valid result when rounded to $N' - 5$ bits with a bias of 3/4.

On the IBM 370, $N' - 5$ bits in double precision will suffice ($N' = 56$ and the maximum N is 51 for that format). If one wants to get by with $N' - 4$ and $N' \geq n$ (we have already covered $N' < n$), then scaling β, x, and y by powers of two to make them binary-normalized in the hexadecimal representation will by itself effectively increase the precision of the multiplication to at least $N' - 1$ bits, chopped, as in section 5.1. The final division then chops to at least $N' - 3$ bits, allowing for a correctly-rounded $N' - 4$-bit result.

Finally, although again there is no official need for it, one can squeeze by with $N' - 3$ bits (the theoretical maximum for hexadecimal). This involves adjusting the product of the rounded β by x to have exactly one leading binary 0. When this is divided by a binary-normalized y, the leading hexadecimal digit will have to be either 8 or 4—at least $N' - 1$ bits of precision, chopped. The prior analysis on binary arithmetic with chopping operators now applies, with $N' - 1$ in place of N'.

Algorithms 6 and 7 summarize the analysis of fixed-point divisions that yield floating-point results.

5.3 Implementation notes on Algorithms 5–7

Arithmetic conforming to IEEE 754 meets the requirements of these algorithms with either single-precision format ($N' = 24$, which may be used for types specified with up to 6 decimal digits of accuracy or a maximum $N = 21$) or double-precision format ($N' = 53$, good for up to 15 decimal digits or $N = 51$). IBM 370 hexadecimal arithmetic also meets the requirements with either single-precision ($N' = 24$ and $N = 21$) or double-precision ($N' = 56$ and $N = 51$) format. VAX arithmetic meets the requirements with F_floating format ($N' = 24$ and $N = 21$), with D_floating format ($N' = 56$ and $N = 31$), and, for floating-point types requiring fewer than 15 decimal digits, with G_floating format ($N' = 53$ and $N = 51$).

The case of D_floating format on the VAX is a fluke; the current Ada definition only allows this format to be used to represent types with up to 9 decimal digits of accuracy, giving it an enormous number of extra bits. As a result, one can use D_floating format for doing conversions to G_floating format for cases such as $N = 51$, where $N' = 53$ is insufficient. One must simply take care to re-scale the intermediate results to avoid overflowing D_floating's smaller exponent range. With this caveat, Algorithms 5–7 apply to all Ada floating-point types representable in VAX G_floating format.

The biased rounding operation, $\text{round}_\lambda(\gamma, M)$, is easily implemented if there is an integer $l \geq 0$ such that $\lambda = l \cdot 2^{M-N'}$, as in the algorithms used in this paper. If on a machine with arithmetic that chops, the operation (for $\gamma \neq 0$) amounts to adding $\text{sign}(\gamma) \cdot l \cdot 2^{\text{logb}(\gamma)-N'+1}$ to γ and then masking off the last $N' - M$ bits if the addition does not increment the exponent, and

otherwise the last $N' - M - 1$ bits. It may, however, be faster to manipulate the representation as integers.

The algorithms use a number of other operations that are carefully disguised in the harmless-looking mathematical notation used to express them. These include

- scaling and shifting, represented as multiplications and divisions by powers of 2;
- conversions, represented implicitly by, for example, the application of floating-point operations to integers;
- assorted other operations, notably $\mathrm{logb}(\cdot)$, $\mathrm{adj}^h(\cdot)$ and $\mathrm{round}^h(\cdot,\cdot)$.

Realization of some of these operations will involve significant code sequences. However, I will not elaborate on them here, because they are typically representation-dependent and in any case present no conceptual difficulties.

6 Looser Accuracy and Range Requirements

If a fixed-point type declaration requires a representation type with only $n' < n$ significant binary digits, a compiler may (and often will) choose to use a representation whose *small* value is less than that required by some power of two. For example, consider the following Ada declaration on a machine with 32-bit words.

```
type T16 is delta 1.0 range -2**16 .. 2**16;
```

The type *T16* requires only 17 bits for its representation (the semantics of Ada do not require that 2^{16} be representable). It will often be represented by integers in the full range, -2^{31} to $2^{31} - 1$, with the value 1 being represented by 2^{15}. For this choice of representation, results of multiplications and divisions that are coerced to type *T16* need only be accurate to within 2^{15} rather than 1. Alternatively, the type *T16* can be represented by 32-bit integers in the range -2^{16} to $2^{16} - 1$. In this case, the legitimate intermediate values produced by Algorithms 1 and 2 have a restricted range.

Of course, the Algorithms 1 and 2 will work for both of these choices of representation. It is reasonable to inquire, however, whether these less stringent requirements on the accuracy of the final result or on the range of intermediate results might be exploited to simplify the algorithms. To a certain extent, they can.

When x and y are restricted to particular signs (non-positive or non-negative), tests of $\mathrm{sign}(x)$ and $\mathrm{sign}(xy)$ may be eliminated. When the

operands' ranges are sufficiently restricted, it is sometimes possible to substitute single-length additions, subtractions, or comparisons for the double-length operations generally required. In operations that require left shifts, the shifting may be transferred to the operands and done using single-length shifts, again depending on the operands' ranges. This is advantageous when shifting of double-length quantities is relatively expensive. All of these simplifications are sufficiently straightforward that I won't go into details.

When the representation chosen for the result type has k "extra bits" on the right, so that *small* is represented by 2^k, one cannot, unfortunately, get away with the obvious adjustments to equations (1) and (2):

$$|r - \alpha xy| < 2^k \tag{8}$$

$$|s - \beta x/y| < 2^k. \tag{9}$$

These conditions are necessary but not sufficient to guarantee that r and s end up in the same minimal safe intervals as αxy and $\beta x/y$. However, if r and s satisfy these conditions and are safe numbers of the result type themselves, then they are valid results, since the conditions then preclude there being any other safe numbers between r or s and the mathematically correct values.

Algorithms 8 and 9 are modifications of Algorithms 1 and 2 suitable for use with $k > 0$. These algorithms use the operation floor(q, m), which is the result of rounding the integer q down (toward $-\infty$) to the nearest multiple of 2^m. Both produce valid safe numbers either of the desired result type or of a finer type (i.e., whose *small* is smaller by a power of 2 from that of the result type); such results are valid by Fact 6).

6.1 Coarse multiplication: Algorithm 8

Let $\hat{r} = \text{shift}(x * y, m) \div a$ and

$$\hat{\delta} = \hat{r} - \alpha xy = \epsilon\alpha xy - \rho_2/a - \rho_1,$$

with ρ_1 and ρ_2 as previously. Let $\delta = r - \alpha xy$.

If $\epsilon \geq 0$, then $-1 < \hat{\delta} < 1$ when $xy \geq 0$ and $-2 < \hat{\delta} < 1$ when $xy < 0$. Since for this case the algorithm computes $r = \text{floor}(\hat{r} + 1,\ 1)$, we have $-1 < \delta < 2$ when $xy \geq 0$ and $-2 < \delta < 2$ with $xy < 0$, using the fact that $0 \leq z - \text{floor}(z, m) < 2^m - 1$ for $m > 0$. Since the floor operation here yields a safe number for a type with *small* represented by 2 and $|\delta| < 2$, the resulting value of r is valid, by Fact 6).

If $k > 1$, and $\epsilon < 0$, then $-2 < \hat{\delta} < 0$ when $xy \geq 0$ and $-1 < \hat{\delta} < 2$ when $xy < 0$. The algorithm computes $r = \text{floor}(\hat{r} + 2,\ 2)$, giving $-3 < \delta < 2$ for $xy \geq 0$ and $-2 < \delta < 4$ for $xy < 0$. Since $|\delta| < 4$ and r is a safe number

To find an r satisfying (1) or a safe number r satisfying (8), for $k > 0$, $-2^n \le \alpha xy \le 2^n - 1$.

```
-- Assume that 2^k ξ is the normal form of round(1/α, n).
a := 2^n ξ;  m := n − k;
if   ε ≥ 0 and (m ≥ 0 or ε < 2^{−n} (a−1+2^m)/a) then
        -- This case is simply taken from Algorithm 1.
        r := shift(x * y, m);
elsif ε ≥ 0 then
        r := floor(shift(x * y, m) ÷ a +_s 1, 1);
elsif k > 1 then
        r := floor(shift(x * y, m) ÷ a +_s 2, 2);
elsif k = 1 then
        r := floor(shift(x * y, m) ÷ a +_s sign_+(xy), 1);
end if;
```

Algorithm 8: Coarse multiplication of fixed-point quantities, yielding a fixed-point result whose *small* is represented by 2^k.

for a type with *small* represented by 4, it is correct. If we instead compute $r = \text{floor}(\hat{r} + \text{sign}_+(xy),\ 1)$, we get $-2 < \delta < 1$ for $xy \ge 0$ and $-2 < \delta < 2$ when $xy < 0$. Again, r is a safe number for a *small* of 2 and $|\delta| < 2$. This result is therefore valid when $k = 1$.

6.2 Coarse division: Algorithm 9

Let $\hat{s} = \text{shift}(b * x, m) \div y$ and

$$\hat{\delta} = \hat{s} - \beta x/y = \epsilon\beta x/y - \rho_2/a - \rho_1,$$

Let $\delta = s - \beta x/y$, for the final value of s.

Consider first $\epsilon \ge 0$. If $m \ge 0$ or $x \ge 0$ (and therefore $bx \ge 0$), then $-1 < \hat{\delta} < 1$, and $\hat{s}$ must therefore be a correct result with no further modification. Otherwise, we have $-1 < \hat{\delta} < 2$ for $xy \ge 0$ and $-2 < \hat{\delta} < 1$ for $xy < 0$. Computing $s = \text{floor}(\hat{s} + 2, 2)$ gives $-2 < \delta < 4$ for $xy \ge 0$ and $-3 < \delta < 3$ for $xy < 0$, so this is a proper safe number for $k > 1$. Otherwise, computing $s = \text{floor}(\hat{s} -_s \text{sign}_-(xy),\ 1)$ gives bounds of $-2 < \delta < 2$ for all values of $\text{sign}(xy)$.

Now take $\epsilon < 0$. We get bounds of $-2 < \hat{\delta} < 1$ for $xy \ge 0$ and $-1 < \hat{\delta} < 2$ for $xy < 0$. The computation $s = \text{floor}(\hat{s} + 2,\ 2)$ gives $-3 < \delta < 3$ for $xy \ge 0$ and $-2 < \delta < 4$ for $xy < 0$, which is again acceptable if $k > 1$. Otherwise, computing $s = \text{floor}(\hat{s} +_s \text{sign}_+(xy),\ 1)$ gives bounds of $-2 < \delta < 2$ for $xy \ge 0$ and $-2 < \delta < 2$ for all xy, which is acceptable for $k = 1$.

To find an s satisfying (2) or a safe number s satisfying (9) for $k > 0$, $-2^n \le \beta x/y \le 2^n - 1$.

```
-- Assume that 2^k ξ is the normal form of round(β, n).
b := 2^n ξ;  m := k − n;
s := shift(b * x,m) ÷ y;
if  ε ≥ 0 and (m ≥ 0 or x ≥ 0) then
        null;
orif k > 1 then
        s := floor(s +_s 2, 2);
orif k = 1 and ε ≥ 0 then
        s := floor(s −_s sign_(xy), 1);
orif k = 1 and ε < 0 then
        s := floor(s +_s sign_+(xy), 1);
end if;
```

Algorithm 9: Coarse division of fixed-point quantities, yielding a fixed-point result whose *small* is represented by 2^k.

7 Concluding Remarks

The semantics of Ada fixed-point operations pose some problems for the would-be implementor who wishes to provide a complete set of Ada representation clauses. However, as we have seen, the costs of providing correct implementations of fixed-point multiplication in the presence of arbitrary rational scaling factors are not particularly large for the case of fixed-point results. In the worst case, fixed-point multiplication with fixed-point result (Algorithm 1) requires one multiplication, one division, one shift, and two double-length additions. Fixed-point division with fixed-point result (Algorithm 2) is the same, with a single-length addition substituted for one double-length addition. When the representation type is left-justified in a larger word, the cost of fixed-point multiplication or division with a fixed-point result (Algorithms 8 and 9) can be improved slightly by substituting a single-length addition and a masking operation for the double-length additions.

When results of an integer type are required—in which case Ada semantics requires rounding to half a unit in the last place—the compiler's case analysis becomes more complex and the algorithms generally more costly. In the worst case, fixed-point multiplication with nearest integer result (Algorithm 3) requires two divisions—one with remainder—four multiplications, one shift, and three double-length additions. Fixed-point division with nearest integer result (Algorithm 4) requires in the worst case two divisions, three

multiplications, a shift, and three double-length additions.

Further, for either Algorithm 3 or 4 to apply, α or β must be precisely expressible in the form $2^m a/b$ for single-length a and b. I suggest that this is not an unreasonable implementation restriction. It is probably rare really to need true rounding to nearest for a multiplication or division where the conversion factor is not expressible in the necessary form. If true rounding to nearest is not actually needed, it is easy (but requires a bit more writing) to specify that one wants integer values that are rounded to one unit in the last place.

Usually, floating-point results of multiplication require an integer multiplication, a floating-point multiplication, a conversion from double-length integer to floating-point, and assorted twiddling. Division requires a floating-point multiply, and floating-point or integer division, and more assorted twiddling. The necessary operations are available on reasonably-behaved hardware—this paper considered machines like the VAX, the IBM 370, and anything conforming to IEEE 754. The operations require that the floating-point computation type have a few extra bits than required for the safe numbers (two for IEEE Standard machines, three for binary-radix machines that round, three or four for binary-radix machines that chop (depending on whether integers can be represented exactly). The specific machines considered will always have these extra bits, and similar architectures may also have them. I have not considered less well-behaved floating-point architectures as a matter of policy.

There are a number of possible improvements to the results presented. First, I have quoted worst-case results; it is unclear what the average case costs are for any of these algorithms. Second, these algorithms all take full advantage of the indeterminate roundings allowed by the Ada semantics. Their precise rounding behaviors, which can be important in sufficiently delicate numerical codes, cannot be described with anything approaching the cleanliness of IEEE 754. Finally, I have paid no particular attention to the effects of supplying operands that are outside the domains specified for the inputs. Although this is valid according to the strict Ada semantics, where not even division by 0 need cause an exception for non-integer scalar types, a production implementation will probably want to take more care.

Finally, several of the procedures—in particular Algorithms 5–9—all take some pains to remove precision from the result in order not to violate rules on safe numbers. The reader who concludes that this indicates a problem with the Brown-model style semantics of Ada's real arithmetic will get no argument from me.

Acknowledgement. My thanks to Jean-Marc Chebat, Terry Froggatt, and William Kahan for their comments and assistance. Any remaining errors

are, of course, my own.

References

1. Terry Froggatt, "Fixed-point conversion, multiplication, & division, in Ada(R)." *Ada Letters* 7, 1 (Jan., Feb. 1987), pp. 71–81.

2. Jean-Pierre Rosen, *Une machine virtuelle pour Ada: le système d'exploitation.* Ph.D. thesis, ENST, 1986. Paris, France.

Appendix: Collected Arithmetic Facts

The following facts are sufficiently simple that they are presented without proof.

Fact 1 $\text{trunc}(U_1/u_2) = U_1/u_2 - \rho_1$, where $0 \leq |\rho_1| \leq \min(|U_1/u_2|, 1 - 1/|u_2|)$ and $\text{sign}(\rho_1) = \text{sign}(U_1 u_2)$. This result therefore applies to $U_1 \div u_2$ when the latter is defined.

Fact 2 $\text{shift}(U,\ m) = 2^m U - \rho_2$, where $0 \leq \rho_2 \leq \min(|2^m U|, 1 - 2^m)$ for $m < 0$, and $\rho_2 = 0$ for $m \geq 0$.

Fact 3 $\text{trunc}((U_1 + \text{sign}(U_1)\lfloor |u_2|/2 \rfloor)/u_2) \in [U_1/u_2 \pm 1/2]$. This follows directly from Fact 1) and provides a way to produce rounded results via integer division.

Fact 4 If $m < 0$ then $\text{shift}(\text{trunc}(U_1/u_2) + 2^{-m-1} + \text{sign\_}(U_1 u_2),\ m) \in [2^m U_1/u_2 \pm 1/2]$. This is somewhat obscure, but follows from Facts 1 and 2.

Fact 5 $\text{trunc}(\text{trunc}(U_1/u_2)/u_3) = \text{trunc}(U_1/u_2 u_3)$, so $(U_1 \div u_2) \div u_3 = \text{trunc}(U_1/(u_2 u_3))$, where defined.

Fact 6 If the safe numbers of type T_1 are a superset of those of T_2, then any semantically valid values for $T_1(P * Q)$ and $T_1(P/Q)$ are also valid for $T_2(P * Q)$ and $T_2(P/Q)$ (i.e., considering just numerical values and ignoring data types). This is true both for fixed-point and floating-point types T_1 and T_2.

Fact 7 If $\gamma(1 + \delta) = \text{round}(\gamma, m)$, then $|\delta| \leq 1/(1 + 2^m)$.