Floats extending Floats

Jeffrey Sarnoff

June 19, 2016

1986

tiny perturbations [can] cause relatively serious errors ...
avoidable only by [doubling the bits] in our computations

tiny perturbations [can] cause relatively serious errors ...
avoidable only by [doubling the bits] in our computations

2004

badly inflated are the costs
of developing and maintaining

high quality floating-point software

without
arithmetic precision
twice as wide

as the given data and desired results.

doubling the bits

arithmetic
twice as wide
as the data and results.

$$time \ \uparrow\downarrow \ bits \ \uparrow\downarrow \ time$$

Researchers found severe numerical inaccuracies when computing planetary orbits over very long time frames. They found only one reliable, accurate way to obtain multi-epochal orbits: compute with double-doubles.

Physicists at the Large Hadron Collider compute scattering amplitudes using Float64s. In numerically unstable regions of phase space, they recompute using double-doubles.

from "High-Precision Computation and Mathematical Physics" David H. Bailey and Jonathan M. Borwein; ©2009
from "Mixed-Precision Cholesky QR .. on Multicore CPU with .. GPUS" I. Yamazaki, S. Tomov, J. Dongarra; ©2015 SIAM

$$time \ \uparrow\downarrow \ bits \ \uparrow\downarrow \ time$$

Researchers found severe numerical inaccuracies when computing planetary orbits over very long time frames. They found only one reliable, accurate way to obtain multi-epochal orbits: compute with double-doubles.

Physicists at the Large Hadron Collider compute scattering amplitudes using Float64s. In numerically unstable regions of phase space, they recompute using double-doubles.

float type	single core, no GPU	multicore + multiple GPUS
Float64	1.0	1.0
Float128	8.5	1.4
efficiency	12%	70%

relative speed with double-double as Float128
mixed-precision Cholesky QR factorization

from "High-Precision Computation and Mathematical Physics" David H. Bailey and Jonathan M. Borwein; ©2009
from "Mixed-Precision Cholesky QR .. on Multicore CPU with .. GPUS" I. Yamazaki, S. Tomov, J. Dongarra; ©2015 SIAM

Others may choose to use our software
as they choose to use our software.

How to get log(3.0) wrong: use log(3.0)

```
accurate_log3 = 1.0986_1228_8668_1096_9139_5245;
obtained_log3 = log(3.0);

1.0986_1228_8668_109 8   # accurate
1.0986_1228_8668_109 6   # obtained
```

How to get log(3.0) wrong: use log(3.0)

```
accurate_log3 = 1.0986_1228_8668_1096_9139_5245;
obtained_log3 = log(3.0);
```

```
1.0986_1228_8668_109 8   # accurate
1.0986_1228_8668_109 6   # obtained
```

How to get log(3.0)?

```
using CRlibm   # https://github.com/dpsanders/CRlibm.jl

log(3.0, RoundNearest) == accurate_log3

# replace log() with CRlibm.log()
Base.log{T<:Float64}(x::T) = log(x, RoundNearest);

log(3.0) == accurate_log3
```

CRlibm has log, exp, sin, cos, tan, asin, acos, atan, sinh, cosh; is that sufficient coverage?
Float64 functions run 1.5-3.0x faster ... just round over the significand's final digits.

Another way to see log(3.0)'s value: round log(3.0).

```
function clarify(x::AbstractFloat)         # simplified, yet surprisingly helpful
    roundto = -trunc(Int, log10(eps(x)*2)) # round away a few least sig digits
    round(x, roundto, 10)                  # to cull possibly misleading digits
end
```

Another way to see log(3.0)'s value: round log(3.0).

```
function clarify(x::AbstractFloat)           # simplified, yet surprisingly helpful
    roundto = -trunc(Int, log10(eps(x)*2)) # round away a few least sig digits
    round(x, roundto, 10)                  # to cull possibly misleading digits
end
```

```
1.0986_1228_8668_1098          # accurate
1.0986_1228_8668_1096          # obtained
1.0986_1228_8668_1             # clarify( accurate )
1.0986_1228_8668_1             # clarify( obtained )
```

Another way to see log(3.0)'s value: round log(3.0).

```
function clarify(x::AbstractFloat)          # simplified, yet surprisingly helpful
    roundto = -trunc(Int, log10(eps(x)*2)) # round away a few least sig digits
    round(x, roundto, 10)                   # to cull possibly misleading digits
end

1.0986_1228_8668_ 1098         # accurate
1.0986_1228_8668_ 1096         # obtained
1.0986_1228_8668_ 1            # clarify( accurate )
1.0986_1228_8668_ 1            # clarify( obtained )


# a difficult case

beta63 = Float64( beta(big(6), big(3)) );
beta63fp = beta(6.0, 3.0);

0.0059_5238_0952_38 09_52      # beta63
0.0059_5238_0952_38 09_49      # beta63fp
0.0059_5238_0952_38 1          # clarify( beta63 )
0.0059_5238_0952_38 1          # clarify( beta63fp )
```

BigFloat or largely adrift?

BigFloat wraps the Multiple Precision Floating-Point Reliable Library (MPFR)

The precision of a BigFlaot variable is the exact number of bits used for its **significand** including the hidden bit, and the result is correctly rounded.

setprecision(53) and BigFloat works like Float64

setprecision(24) and BigFloat works like Float32

BigFloat or largely adrift?

BigFloat wraps the Multiple Precision Floating-Point Reliable Library (MPFR)

The precision of a BigFlaot variable is the exact number of bits used for its **significand** including the hidden bit, and the result is correctly rounded.

setprecision(53) and BigFloat works like Float64

setprecision(24) and BigFloat works like Float32

- BigFloat **exponents** have a huge dynamic range: ±1_388_255_822_130_839_284
  (1 million million million or 1 billion billion)
- many functions are supported
  (acos, besselj, cbrt, div, exp, floor, gamma, hypot .. zeta)

BigFloat or largely adrift?

BigFloat wraps the Multiple Precision Floating-Point Reliable Library (MPFR)

The precision of a BigFlaot variable is the exact number of bits used for its **significand** including the hidden bit, and the result is correctly rounded.

setprecision(53) and BigFloat works like Float64

setprecision(24) and BigFloat works like Float32

- BigFloat **exponents** have a huge dynamic range: ±1_388_255_822_130_839_284
  (1 million million million or 1 billion billion)
- many functions are supported
  (acos, besselj, cbrt, div, exp, floor, gamma, hypot .. zeta)

- go elsewhere to go fast
- interconverting values with other applications involves care
- the value you think is present may not be the value presented

BigFloat, the ice cream soda of numeric types

```
1.0/7.0                                           # 1/7 = 0.142857 (repeating)
0.142857_142857_14285
setprecision(58)                                  # like Float64 +5 sig bits

# these are useful ways
big(1)/big(7) == convert(BigFloat, 1//7),         # when parsing a string
parse(BigFloat,"0.142857142857142857142857")      # use more digits than reqd
true, 0.142857_142857_142857
```

BigFloat, the ice cream soda of numeric types

```
1.0/7.0                                           # 1/7 = 0.142857 (repeating)
0.142857_142857_14285
setprecision(58)                                  # like Float64 +5 sig bits

# these are useful ways
big(1)/big(7) == convert(BigFloat, 1//7),         # when parsing a string
parse(BigFloat,"0.142857142857142857142857")      # use more digits than reqd
true, 0.142857_142857_142857

# these are ways best avoided
convert(BigFloat, 1/7), parse(BigFloat, string(1/7))
0.142857_142857_1428 492 , 0.142857_142857_1428 501

Float64( convert(BigFloat, 1/7) ), Float64( parse(BigFloat, string(1/7)) )
0.142857_142857_142857, 0.142857_142857_142857  # they still differ
```

BigFloat, the ice cream soda of numeric types

```
1.0/7.0                                        # 1/7 = 0.142857 (repeating)
0.142857_142857_14285
setprecision(58)                               # like Float64 +5 sig bits

# these are useful ways
big(1)/big(7) == convert(BigFloat, 1//7),      # when parsing a string
parse(BigFloat,"0.142857142857142857142857")   # use more digits than reqd
true, 0.142857_142857_142857

# these are ways best avoided
convert(BigFloat, 1/7), parse(BigFloat, string(1/7))
0.142857_142857_1428 492 , 0.142857_142857_1428 501

Float64( convert(BigFloat, 1/7) ), Float64( parse(BigFloat, string(1/7)) )
0.142857_142857_142857, 0.142857_142857_142857  # they still differ

setprecision(180);                      # let's better see what is going on

parse(BigFloat,"0.142857142857142857")  # many zeros, tracks string's value
0.142857_142857_142850_00000_00000_00000_00000_00000_00000_00000_ 002

big(0.14285714285714285)                # ends 500..0, a veridical expansion
0.142857_142857_142849_21269_26812_48881_85411_69166_56494_14062_ 500
```

Errorfree Transformations

An errorfree transformation is as accurate as it is precise; and that is very helpful.

An errorfree transformation is not free from all error. It is free of spurious error.

Errorfree Transformations

An errorfree transformation is as accurate as it is precise; and that is very helpful.

An errorfree transformation is not free from all error. It is free of spurious error.

A **precisely accurate** value is an approximation that is accurate to the precision used.
A value is **precisely accurate** when adjacent values are not nearer the true value.

Errorfree Transformations

An errorfree transformation is as accurate as it is precise; and that is very helpful.

An errorfree transformation is not free from all error. It is free of spurious error.

A **precisely accurate** value is an approximation that is accurate to the precision used.
A value is **precisely accurate** when adjacent values are not nearer the true value.

An errorfree transformation is an algorithm that offers two precisely accurate values:

the ordinary floating point result and an approximation of the residual value;
and these two values are non-overlapping.

$$(\,\boldsymbol{x}, \boldsymbol{y}\,) \leftarrow \mathrm{eft}(\,\mathrm{a}, \mathrm{b}\,) \quad \Longrightarrow \quad \mathrm{eft}(\,\mathrm{a}, \mathrm{b}\,) \equiv \underbrace{\boldsymbol{x} + \boldsymbol{y}}_{\text{by math}} \;\wedge\; \underbrace{\boldsymbol{x} \oplus \boldsymbol{y} == \boldsymbol{x}}_{\text{by fpu}}$$

Errorfree Transformations

```julia
typealias SysFloat Union{Float64,Float32}

function eftAddGTE{T<:SysFloat}(a::T, b::T)
  @assert abs(a) >= abs(b)        # the GTE in eftAddGTE
  sum = a + b
  implicit_b = sum - a
  residuum   = (b - implicit_b)   # what is 'leftover' and usually lost
  sum, residuum
end
```

Errorfree Transformations

```
typealias SysFloat Union{Float64,Float32}

function eftAddGTE{T<:SysFloat}(a::T, b::T)
  @assert abs(a) >= abs(b)         # the GTE in eftAddGTE
  sum = a + b
  implicit_b = sum - a
  residuum   = (b - implicit_b)    # what is 'leftover' and usually lost
  sum, residuum
end

function eftAdd{T<:SysFloat}(a::T, b::T)
  sum = a + b
  implicit_b = sum - a
  residuum   = (b - implicit_b) + (a - (sum - implicit_b))
             # ordered_residuum + (a - ((a+b)-implicit_b))
             # ordered_residuum +   antiordered_residuum    one is 0.0
  sum, residuum
end
```

Errorfree Transformations

```
typealias SysFloat Union{Float64,Float32}

function eftAddGTE{T<:SysFloat}(a::T, b::T)
  @assert abs(a) >= abs(b)        # the GTE in eftAddGTE
  sum = a + b
  implicit_b = sum - a
  residuum   = (b - implicit_b)   # what is 'leftover' and usually lost
  sum, residuum
end

function eftAdd{T<:SysFloat}(a::T, b::T)
  sum = a + b
  implicit_b = sum - a
  residuum   = (b - implicit_b) + (a - (sum - implicit_b))
             # ordered_residuum + (a - ((a+b)-implicit_b))
             # ordered_residuum +   antiordered_residuum    one is 0.0
  sum, residuum
end
```

eftAdd, eftAddGTE, eftSub, eftMul, eftFMA, accSqrt, accDiv, accHypot

FMA enhanced Errorfree Transformation

All EFT are algorithmic maps of floating point activity onto a well-spread sum.
We obtain the usual float result and a residual to add for a twice precise result.

FMA enhanced Errorfree Transformation

All EFT are algorithmic maps of floating point activity onto a well-spread sum.
We obtain the usual float result and a residual to add for a twice precise result.

many EFTS go from lethargic to energetic with the use of FMA

```
function eftMul{T<:SysFloat}(a::T, b::T)
    hi = a * b
    lo = fma(a, b, -hi)        #  (a × b) - (a ⊗ b)
                               #  lo is the value that is usually lost
    hi, lo
end                            #  2 flops (without fma: 17 flops)
```

FMA enhanced Errorfree Transformation

All EFT are algorithmic maps of floating point activity onto a well-spread sum.
We obtain the usual float result and a residual to add for a twice precise result.

many EFTS go from lethargic to energetic with the use of FMA

```
function eftMul{T<:SysFloat}(a::T, b::T)
    hi = a * b
    lo = fma(a, b, -hi)        #  (a × b) - (a ⊗ b)
                               #  lo is the value that is usually lost
    hi, lo
end                            #  2 flops (without fma: 17 flops)

function eftMulAs3{T<:SysFloat}(a::T, b::T, c::T)
     abHi , abLo   = eftMul(a, b)
     hi   , mid    = eftMul(abHi, c)

     lo            = abLo * c # drop the least sig (4th) part

     hi, mid, lo
end                           # 5 flops (without fma: 35 flops)
```

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.

When errorfree ops do the compensating, some computations are multifold accurate.

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.
When errorfree ops do the compensating, some computations are multifold accurate.

Precisely accurate values obtain when each next bit portrays more of the same true value.
To be precisely accurate, working precision and algorithmic valuation must collaborate.

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.
When errorfree ops do the compensating, some computations are multifold accurate.

Precisely accurate values obtain when each next bit portrays more of the same true value.
To be precisely accurate, working precision and algorithmic valuation must collaborate.

A compensated (**csd**) calculation is an algorithm that offers an n-fold precise accuracy:

$$x \leftarrow \mathbf{csdSum}(\,\mathbf{a} :: \mathrm{Float64}\,) \quad \Longrightarrow \quad \mathbf{csdSum}(\,\mathbf{a}\,) \equiv \mathrm{Float64}(\,a_1 + \ldots + a_n\,)$$

The result is as if computed using n-fold precision
and then rounded to the common precision before use.

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.
When errorfree ops do the compensating, some computations are multifold accurate.

Precisely accurate values obtain when each next bit portrays more of the same true value.
To be precisely accurate, working precision and algorithmic valuation must collaborate.

A compensated (**csd**) calculation is an algorithm that offers an n-fold precise accuracy:

$$x \leftarrow \textbf{csdSum}(\, \mathbf{a} :: \text{Float64}\,) \quad \Longrightarrow \quad \textbf{csdSum}(\, \mathbf{a}\,) \equiv \text{Float64}(\, a_1 + \ldots + a_n\,)$$

The result is as if computed using n-fold precision
and then rounded to the common precision before use.

csdSum, csdProd, csdHypot, csdPow, csdDot, csdHorner, csdSumNx, csdDotNx, csdHornerNx

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.
When errorfree ops do the compensating, some computations are multifold accurate.

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.

When errorfree ops do the compensating, some computations are multifold accurate.

csdDot(x,y) has the same type as dot(x,y) and stores a more accurate value:
it calculates at twice working precision then rounds to working precision.

Compensated Calculation

A compensated calculation is as a well-balanced balance, free from introduced error.
When errorfree ops do the compensating, some computations are multifold accurate.

csdDot(x,y) has the same type as dot(x,y) and stores a more accurate value:
it calculates at twice working precision then rounds to working precision.

```
function csdDot{T<:SysFloat}(x::Vector{T}, y::Vector{T})
  @assert (length(x)>0) && (length(x) == length(y))

  sHi, compensation = eftMul(x[1], y[1])
  for i in 2:length(x)
      pHi, pLo = eftMul(x[i], y[i])   # errorfree for ith pair multiply
      sHi, sLo = eftAdd(sHi, pHi)     # update sum at twice working precision
      compensation += pLo + sLo       # accumulate most of the residual value
  end

  sHi + compensation                  # as if rounded from 2x significant bits
end
```

$$X^2 \pm Y^2, \quad B^2 \pm A^*C, \quad A^*B \pm C^*D$$

William Kahan's method, relative accuracy is ± 2 bits

```
function ad_minus_bc{T<:SysFloat}(a::T, b::T, c::T, d::T)
    bcHi  = b*c                  # precisely accurate
    bcLo  = fma(-b, c,  bcHi)   # precisely accurate

    hi    = fma( a, d, -bcHi)   # approximates a*d - b*c
    hi - bcLo                    # compensated approximation
end

ab_minus_cd{T<:SysFloat}(a::T, b::T, c::T, d::T) = ad_minus_bc(a,c,d,b)
```

$$X^2 \pm Y^2, \quad B^2 \pm A^*C, \quad A^*B \pm C^*D$$

William Kahan's method, relative accuracy is ± 2 bits

```
function ad_minus_bc{T<:SysFloat}(a::T, b::T, c::T, d::T)
    bcHi  = b*c                  # precisely accurate
    bcLo  = fma(-b, c,  bcHi)    # precisely accurate

    hi    = fma( a, d, -bcHi)    # approximates a*d - b*c
    hi - bcLo                    # compensated approximation
end

ab_minus_cd{T<:SysFloat}(a::T, b::T, c::T, d::T) = ad_minus_bc(a,c,d,b)


function cross3D{T<:SysFloat}(a::Vector{T}, b::Vector{T})
    a1, a2, a3 = a[1], a[2], a[3]
    b1, b2, b3 = b[1], b[2], b[3]

    x = ad_minus_bc(a2, a3, b2, b3)    # a2*b3 - a3*b2
    y = ad_minus_bc(a3, a1, b3, b1)    # a3*b1 - a1*b3
    z = ad_minus_bc(a1, a2, b1, b2)    # a1*b2 - a2*b1

    [x,y,z]
end
```

Extending Floats

```
import Base: convert, promote_rule
typealias SysFloat Union{Float64, Float32}

immutable FFloat{T<:SysFloat} <: Real          # call me FloatFloat
    hi::T                            # high order part, ordinary precision
    lo::T                            # low  order part, extended precision

    FFloat{T}(hi::T, lo::T) =        #  magnitude of hi >= magnitude of lo
        ifelse(abs(hi) > abs(lo), new(hi,lo) : new(lo,hi))
end
```

Extending Floats

```
import Base: convert, promote_rule
typealias SysFloat Union{Float64, Float32}

immutable FFloat{T<:SysFloat} <: Real          # call me FloatFloat
    hi::T                                # high order part, ordinary precision
    lo::T                                # low  order part, extended precision

    FFloat{T}(hi::T, lo::T) =       #  magnitude of hi >= magnitude of lo
        ifelse(abs(hi) > abs(lo), new(hi,lo) : new(lo,hi))
end

#  matching to the internal constructor to allow implicit parameterization
FFloat{T<:SysFloat}(hi::T, lo::T)  = FFloat{T}(hi, lo)

#  define explicit conversions for faster immutable type construction
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T, lo::T) = FFloat(hi, lo)
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T)        = FFloat(hi, zero(T))
```

Extending Floats

```
import Base: convert, promote_rule
typealias SysFloat Union{Float64, Float32}

immutable FFloat{T<:SysFloat} <: Real          # call me FloatFloat
    hi::T                              # high order part, ordinary precision
    lo::T                              # low  order part, extended precision

    FFloat{T}(hi::T, lo::T) =       #  magnitude of hi >= magnitude of lo
        ifelse(abs(hi) > abs(lo), new(hi,lo) : new(lo,hi))
end

#  matching to the internal constructor to allow implicit parameterization
FFloat{T<:SysFloat}(hi::T, lo::T)  = FFloat{T}(hi, lo)

#  define explicit conversions for faster immutable type construction
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T, lo::T) = FFloat(hi, lo)
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T)        = FFloat(hi, zero(T))

#  interrelate this type with the system floats
convert{T<:SysFloat}(::Type{T}, x::FFloat{T}) = x.hi
promote_rule{T<:SysFloat}(::Type{T}, ::Type{FFloat{T}}) = FFloat{T}
```

Extending Floats

```
import Base: convert, promote_rule
typealias SysFloat Union{Float64, Float32}

immutable FFloat{T<:SysFloat} <: Real         # call me FloatFloat
    hi::T                                # high order part, ordinary precision
    lo::T                                # low  order part, extended precision

    FFloat{T}(hi::T, lo::T) =       #  magnitude of hi >= magnitude of lo
        ifelse(abs(hi) > abs(lo), new(hi,lo) : new(lo,hi))
end

#  matching to the internal constructor to allow implicit parameterization
FFloat{T<:SysFloat}(hi::T, lo::T)  = FFloat{T}(hi, lo)

#  define explicit conversions for faster immutable type construction
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T, lo::T) = FFloat(hi, lo)
convert{T<:SysFloat}(::Type{FFloat{T}}, hi::T)        = FFloat(hi, zero(T))

#  interrelate this type with the system floats
convert{T<:SysFloat}(::Type{T}, x::FFloat{T}) = x.hi
promote_rule{T<:SysFloat}(::Type{T}, ::Type{FFloat{T}}) = FFloat{T}

#  handle other likely inputs
FFloat{R<:Real}(hi::R) = FFloat( float( hi ), float( hi-float(hi) ) )
FFloat{R1<:Real,R2<:Real}(hi::R1, lo::R2) = FFloat(float(hi), float(lo))
```

The Minimal Substrate

```
import Base: (==), hash, sizeof, string, show, copy

                                           clear two most significant bits
const hash_ff_lo = (UInt === UInt64) ? 0x086540d7a5325bc3 : 0x5acda43c
const hash_0_ff_lo = hash(zero(UInt), hash_ff_lo)

hash{T<:SysFloat}(z::FFloat{T}, h::UInt) =                # (==) → same hash
    hash(z.hi, h $ hash(z.lo, hash_ff_lo) $ hash_0_ff_lo)

(==){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = (a.hi == b.hi) & (a.lo == b.lo)
```

The Minimal Substrate

```
import Base: (==), hash, sizeof, string, show, copy

                                          clear two most significant bits
const hash_ff_lo = (UInt === UInt64) ? 0x086540d7a5325bc3 : 0x5acda43c
const hash_0_ff_lo = hash(zero(UInt), hash_ff_lo)

hash{T<:SysFloat}(z::FFloat{T}, h::UInt) =              # (==) → same hash
    hash(z.hi, h $ hash(z.lo, hash_ff_lo) $ hash_0_ff_lo)

(==){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = (a.hi == b.hi) & (a.lo == b.lo)

sizeof{T<:SysFloat}(z::FFloat{T}) = sizeof(z.hi) << 1

copy{T<:SysFloat}(x::FFloat{T}) = FFloat{T}(x.hi, x.lo)

# show{T}(x::T) should look much the same as if x were typed into the REPL

string{T<:SysFloat}(x::FFloat{T}) =
    string( "FFloat(",  x.hi,  ", ",  x.lo,  ")" )

show{T<:SysFloat}(io::IO, x::FFloat{T}) = print(io, string(x))
```

Comparison and Ordering

```
import Base: isequal, isless, (==), (!=), (<=), (>), (<), (>=)

(==){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi == b.hi && a.lo == b.lo)

(<=){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi < b.hi || ((a.hi==b.hi) & (a.lo <= b.lo)))

( <){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi < b.hi || ((a.hi==b.hi) & (a.lo <  b.lo)))

(!=){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a==b)
( >){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a<=b)
(=>){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a<=b)
```

Comparison and Ordering

```
import Base: isequal, isless, (==), (!=), (<=), (>), (<), (>=)

(==){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi == b.hi && a.lo == b.lo)

(<=){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi < b.hi || ((a.hi==b.hi) & (a.lo <= b.lo)))

( <){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) =
    (a.hi < b.hi || ((a.hi==b.hi) & (a.lo <  b.lo)))

(!=){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a==b)
( >){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a<=b)
(=>){T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = !(a<=b)


isequal{T<:SysFloat}(a::FFloat{T}, b::FFloat{T}) = (==)(hash(a), hash(b))
isless{T<: SysFloat}(a::FFloat{T}, b::FFloat{T}) = (<)(a,b)
```

Proto-Numerics

```
import Base: signbit, sign, abs, isnan, isinf, isfinite, zero, one, eps,
             copysign, flipsign, frexp, ldexp

for (F) in (:signbit, :sign, :isfinite, :isnan, :isinf)
    @eval ($F){T<:SysFloat}(x::FFloat{T})  = ($F)(x.hi)
end

abs{T<:SysFloat}(x::FFloat{T}) =
    signbit(x.hi) ? FFloat{T}(-x.hi, -x.lo) : copy(x)
```

Proto-Numerics

```
import Base: signbit, sign, abs, isnan, isinf, isfinite, zero, one, eps,
             copysign, flipsign, frexp, ldexp

for (F) in (:signbit, :sign, :isfinite, :isnan, :isinf)
    @eval ($F){T<:SysFloat}(x::FFloat{T})  = ($F)(x.hi)
end

abs{T<:SysFloat}(x::FFloat{T}) =
    signbit(x.hi) ? FFloat{T}(-x.hi, -x.lo) : copy(x)

zero{T<:SysFloat}(::Type{FFloat{T}}) = FFloat(0.0, 0.0)  # similarly
zero{T<:SysFloat}(x::FFloat{T})      = zero(T)           # for one()
```

Proto-Numerics

```
import Base: signbit, sign, abs, isnan, isinf, isfinite, zero, one, eps,
          copysign, flipsign, frexp, ldexp

for (F) in (:signbit, :sign, :isfinite, :isnan, :isinf)
    @eval ($F){T<:SysFloat}(x::FFloat{T})  = ($F)(x.hi)
end


abs{T<:SysFloat}(x::FFloat{T}) =
    signbit(x.hi) ? FFloat{T}(-x.hi, -x.lo) : copy(x)

zero{T<:SysFloat}(::Type{FFloat{T}}) = FFloat(0.0, 0.0)  # similarly
zero{T<:SysFloat}(x::FFloat{T})      = zero(T)           # for one()

eps{T<:SysFloat}(::Type{FFloat{T}})  = eps(eps(one(T)))/2

function eps{T<:SysFloat}(x::FFloat{T})
    if x.lo != 0.0
        eps(x.lo)              # the lo part is nonzero
    elseif a.hi != 0.0
        eps(eps(x.hi))*0.5     # the lo part is zero
    else
        eps(FFloat{T})         # the value is zero
    end
end
```

Extending Float Arithmetic

```
function (+){T<:SysFloat}(a::FFloat{T}, b::T)
    hi, lo = eftAdd(a.hi, b)          # errorfree

    lo += a.lo                        # compensation
    hi, lo = eftAddGTE(hi, lo)        # renormalization

    FFloat(hi, lo)                # parameter is implied
end

(+){T<:SysFloat}(a::T, b::FFloat{T}) = (+)(b,a)
```

Extending Float Arithmetic

```
function (+){T<:SysFloat}(a::FFloat{T}, b::T)
    hi, lo = eftAdd(a.hi, b)        # errorfree

    lo += a.lo                      # compensation
    hi, lo = eftAddGTE(hi, lo)      # renormalization

    FFloat(hi, lo)              # parameter is implied
end

(+){T<:SysFloat}(a::T, b::FFloat{T}) = (+)(b,a)

function (*){T<:SysFloat}(a::FFloat{T}, b::FFloat{T})
    hi, lo = eftMul(a.hi,b.hi)      # errorfree transformation
    md   = a.hi * b.lo              #   compensating constituent
    md  += a.lo * b.hi              #   compensating constituents

    lo  +=  md                      # compensation
    hi, lo = eftAddGTE(hi,lo)       # renormalize compensated (*)

    FFloat(hi, lo)
end

(*){T<:SysFloat}(a::FFloat{T}, b::T) = (*)(a, FFloat{T}(b))
(*){T<:SysFloat}(a::T, b::FFloat{T}) = (*)(promote(a,b)...)
```

Introducing ArbFloats

powered by Fredrik Johansson's Arb, through William Hart's Nemo.jl

ArbFloats are intervals (midpoint ±radius)

- values are viewed as floating point values: round(underlying, **n**)
- **n** s.t. round(midpoint+radius, **n**) == round(midpoint-radius, **n**)

Introducing ArbFloats

powered by Fredrik Johansson's Arb, through William Hart's Nemo.jl

ArbFloats are intervals (midpoint ±radius)

values are viewed as floating point values: round(underlying, **n**)

n s.t. round(midpoint+radius, **n**) == round(midpoint-radius, **n**)

many functions are supported, including

iszero, ispositive, isinteger, isexact, midpoint, radius

ldexp, hypot, log, exp, (a)sin[h], (a)cos[h], (a)tan[h], atan2

floor, ceil, root, fib, gamma, lgamma, digamma, risingfactorial,

overlap, contains, zeta, agm

Introducing ArbFloats

powered by Fredrik Johansson's Arb, through William Hart's Nemo.jl

ArbFloats are intervals (midpoint ±radius)

- values are viewed as floating point values: round(underlying, **n**)
- **n** s.t. round(midpoint+radius, **n**) == round(midpoint-radius, **n**)

many functions are supported, including

- iszero, ispositive, isinteger, isexact, midpoint, radius
- ldexp, hypot, log, exp, (a)sin[h], (a)cos[h], (a)tan[h], atan2
- floor, ceil, root, fib, gamma, lgamma, digamma, risingfactorial,
- overlap, contains, zeta, agm

ArbFloats are best with data of narrow intervals

- usually, the radius is within a factor of 1.5x..5x of best
- as a guide, the radius may be within $1.5^{N_{ops}}$ of best

ArbFloat significands cover much ground

- significand precision is settable to 7..1200 digits (24..4K bits)
- on 64 bit machines, 35 digit significands use no indirect space

Amiable ArbFloats

```
using ArbFloats
setprecision(ArbFloat, 122)                       # 30 digits quite reliably
```

Amiable ArbFloats

```
using ArbFloats
setprecision(ArbFloat, 122)                          # 30 digits quite reliably

a = gamma(ArbFloat(33)); reciprocal_a = inv(a);
a, reciprocal_a
26313_083693_36935_30167_21801_21600_00000.0      # 1/4 trillion^3
3.80039_075485_47435_92593_67089_27884_1279e-36   # 1/that
```

Amiable ArbFloats

```
using ArbFloats
setprecision(ArbFloat, 122)                         # 30 digits quite reliably

a = gamma(ArbFloat(33)); reciprocal_a = inv(a);
a, reciprocal_a
26313_083693_36935_30167_21801_21600_00000.0      # 1/4 trillion^3
3.80039_075485_47435_92593_67089_27884_1279e-36   # 1/that

recovered_a = inv(reciprocal_a); showall(recovered_a)     # a with fuzz
2.63130_83693_36935_30167_21801_21600_00001e+35 ± 0.5634_9781_99377_656

a ≈ recovered_a
true
```

Amiable ArbFloats

```
using ArbFloats
setprecision(ArbFloat, 122)                          # 30 digits quite reliably

a = gamma(ArbFloat(33)); reciprocal_a = inv(a);
a, reciprocal_a
26313_083693_36935_30167_21801_21600_00000.0         # 1/4 trillion^3
3.80039_075485_47435_92593_67089_27884_1279e-36      # 1/that

recovered_a = inv(reciprocal_a); showall(recovered_a)      # a with fuzz
2.63130_83693_36935_30167_21801_21600_00001e+35 ± 0.5634_9781_99377_656

a ≊ recovered_a
true

e = exp(ArbFloat(1));
bounds(e)
( 2.7182_8182_8459_0452_3536_0287_4713_5266_2 49 ,
  2.7182_8182_8459_0452_3536_0287_4713_5266_2 50  )

showsmart(e)
2.71828182845904523536028747135266 2 5_              # postfixes ~,+,-
```

...

Arb + BigFloat != ArbFloat

set bit precision to get well-behaved digits

digits	25	50	100	150	300	1000	digits
bits	110	175	355	520	1020	3345	bits

BigFloat and Arb

muladd	$5 \propto 8$	$2 \propto 3$	$3 \propto 4$	$1 \propto 1$	muladd
log	$1 \propto 8$	$1 \propto 7$	$1 \propto 7$	$3 \propto 4$	log
zeta	$1 \propto 9$	$1 \propto 75$	$1 \propto 48$	$1 \propto 32$	zeta
bits	125	250	500	4000	bits

BigFloat rounds values correctly, precise numbers that may or may not be known as accurate.
Arb is much faster for 150 digits and rounds to include the accurate value, less precisely.

ArbFloats are performant, mindful and honest.

Floats extending Floats
presented at JuliaCon 2016

by Jeffrey Sarnoff
(2016-Jun-24, Cambridge MA USA)