Reals, Bitvectors, Ordinals, and Ulps: a complete formalization for *Titanic*

Bill Zorn

September 7, 2017

Abstract

Floating point number representations are complicated; we seek to provide a crisp definition of the terms in the title and the relationships between them. Traditionally, floating point numbers are thought of as representing the values of real numbers. In some situations, we find that it makes more sense to impose an ordering on them and treat them like signed integer ordinals, for example when computing units in the last place difference (ulps), or the Posix `nextafter()` function. We formalize the mathematical underpinnings of these relationships as operations on bitvectors.

Work In Progress:
This vessel is not yet seaworthy.

1 Introduction

Floating point is tricky to get right. The idea is simple: represent real numbers with a discrete representation (such as bitvectors) that can be stored and manipulated by a computer. However, the implementation tends to be complex. Sometimes the difference between $\frac{4}{3}$ and 1.3333334 is unimportant, but sometimes it is the difference between the right answer and the wrong answer, or between SAT and UNSAT. The problem is that while they represent familiar, continuous real numbers, their discrete representation makes floating point numbers incredibly specific. If you really want the right answer, you have to be precise down to the last bit. And for most people, a vector of bits like `0b00111111101010101010101010101011` is much less intuitive to think about than the real (rational, even) number $\frac{4}{3}$.

The purpose of this text is to write down in a convient place, once and for all, the meaning of floating point numbers, and formalize the conversions between various bitvector representations that are compatible with the IEEE 754-2008 standard [1] but general enough to be useful in other applications, for example

when devising constraints for SMT solvers. This way, the author (and any other interested implementers) can avoid the trouble of having to write all this down again, or risk getting a bit of it wrong.

2 Notation

In order to be as general as possible, we will try to express relationships and algorithms as simple (integer or real) arithmetic rather than pseudocode which might be more ambiguous to interpret or port to a particular programming language. Since we are often working with bitvectors, it is essential that we define a formal bitvector arithmetic as well.

We treat a bitvector as a nonempty zero-indexed array of bits that each are 0 or 1. We write bitvectors out as the prefix `0b` followed by the contents of the array, from most significant to least significant. A bitvector has size n equal to the number of bits in the array, which must be at least one. The least significant bit is assigned index 0, and the most significant is assigned index $n - 1$.

Bitvectors support the following operations: getting the size, conversion to and from integers, indexing, concatenation and extraction, shifting left and right, and comparison for equality. Every bitvector has a fixed size: $size(\texttt{0b1}) = 1$, while $size(\texttt{0b01}) = 2$. Conversion to and from integers is straightforward and follows the typical binary representation of integers. *uint* is an unsigned conversion, such that $uint(\texttt{0b101}) = 5$, while *sint* is a two's complement signed conversion, such that $sint(\texttt{0b101}) = -3$. Conversion the other way is accomplished with a *bv* constructor, which takes any positive or negative integer value as its first argument and an integer size greater than 0 as its second argument, and returns a bitvector that holds the typical two's complement binary representation of the value, masked to the appropriate size: $bv(1, 2) = \texttt{0b01}$, $bv(-1, 2) = \texttt{0b11} = bv(3, 2)$, and $bv(4, 2) = \texttt{0b00}$.

Indexing is written out using square brackets, and always yields 0 or 1: $\texttt{0b01}[0] = 1$, while $\texttt{0b01}[1] = 0$. Concatenation simply joins two bitvectors together, with the first argument being placed in the more significant bits of the output: $concat(\texttt{0b01}, \texttt{0b101}) = \texttt{0b01101}$. The syntax for extraction is slightly more complex. The first two arguments are the start and end indices, both inclusive, with the first argument being the more significant (i.e. larger). Extraction yields a new bitvector with the bits between these indices: $extract(3, 2, \texttt{0b01101}) = \texttt{0b11}$. For a bitvector x of size n, $extract(n-1, 0, x) = x$ is the identity operation, while for all indices $0 \leq i < n$, $uint(extract(i, i, x)) = x[i]$; the application of *uint* is necessary because extraction yields a bitvector while standard indexing yields an integer.

Shifting is written infix with the right argument always being an integer. An arithmetic right shift is written as `>>`, while a logical one is `>>>`: `0b1101 >> 1` = `0b1110`, `0b1101 >>> 1` = `0b0110`, and `0b1101 << 1` = `0b1010`. Shifts greater than the size are handled normally; they just produce uninteresting results such as `0b1101 >> 8` = `0b1111` and `0b1101 << 257` = `0b0000`. Comparison is only for equality (or inequality), and only allowed between bitvectors of the same size.

To determine ordering, bitvectors can be converted to the appropriate integer representation.

These conventions should translate naturally to most languages and efficient representations, such as C integers. The concatenation and extraction semantics are motivated by and identical to those used in the Rosette language [2].

3 Reals as Bitvectors

A floating point number is a triple of three things: a sign s, an exponent e, and a significand c. s is either 0 or 1 and e is a positive or negative integer. We would like c to represent a fractional value; to do this, we can make it a positive integer and keep around some notion of precision p which records the number of digits in c, as represented in a base b. The real value represented by a number in this representation is given in Equation 1.

$$real(s, e, c, b, p) = (-1)^s \times b^e \times (c \times b^{1-p}) \tag{1}$$

For lack of another widely supported alternative, we will specialize our discussion here to the IEEE 754 (binary) standard. We fix $b = 2$, and commit to representing c as a bitvector C of size p. As we can see in Equation 2, this eliminates the extra terms b and p so that our representation is really a triple.

$$real(s, e, C) = (-1)^s \times 2^e \times (uint(C) \times 2^{1-size(C)}) \tag{2}$$

The IEEE 754 assigns some floating point numbers to represent values other than finite real numbers. Specifically, numbers with greater than some "maximum" e and $c = 0$ represent infinite real numbers of the appropriate sign, and numbers with greater than this "maximum" e but $c \neq 0$ represent something that is not a real number, or NaN, such as the indeterminate result of $\frac{0}{0}$. If we commit to representing e as a bitvector E of size w, then this "maximum" value is $e_{max} = 2^{w-1}-1$, the largest positive integer a bitvector of size w can represent using a two's complement representation. Instead of using two's complement directly, the IEEE 754 standard specifies a biased representation, defining also a minimum exponent $e_{min} = 1 - e_{max}$ and $e = uint(E) - e_{max}$.

Since the values of e and c are already represented as bitvectors, we might as well put s in a 1-bit vector S with $s = uint(S)$. We fully specify the meaning of our mapping from triples of bitvectors to finite real numbers (or one of two infinite real numbers, or a symbol representing something other than a real number) in Equation 3,

$$real(S, E, C) = \begin{cases} NaN & e > e_{max} \wedge c \neq 0 \\ (-1)^s \times \infty & e > e_{max} \wedge c = 0 \\ (-1)^s \times 2^e \times (c \times 2^{1-p}) & e_{min} \leq e \leq e_{max} \\ (-1)^s \times 2^{e_{min}} \times (c \times 2^{1-p}) & e < emin \end{cases} \tag{3}$$

where:

$w = size(E)$

$p = size(C)$

$e_{max} = 2^{w-1} - 1$

$e_{min} = 1 - e_{max}$

$s = uint(S)$

$e = uint(E) - e_{max}$

$c = uint(C)$.

Let us call this the explicit triple of bitvectors or "explicit triple" representation of a floating point number. Except for its neglect of the distinction between quiet and signaling NaNs, this formulation agrees completely with IEEE 754, assuming the ability to identify the correct bitvectors S, E, and C. However, this representation is not as compact: as presented in Equation 3, some real numbers can be represented in multiple ways. For example, the triple $(\texttt{0b0}, \texttt{0b01}, \texttt{0b10})$ represents $(-1)^0 \times 2^0 \times (2 \times 2^{-1}) = 1$. But so does $(\texttt{0b0}, \texttt{0b10}, \texttt{0b01})$, as $(-1)^0 \times 2^1 \times (1 \times 2^{-1}) = 1$ as well. Similarly, having E as 0 or 1 gives an equivalent $e = e_{min}$ due to the fourth case where $e < e_{min}$. And there are $2 \times (2^w - 1)$ ways to represent 0.

Any real number representable with $e > e_{min}$ and $C[p-1] = 0$ (i.e. the high bit of C is 0, or $c < 2^{p-1}$) can be similarly represented by shifting C 1 bit to the left and subtracting 1 from e. Thus each representable number has a canonical representation, where either $C[p-1] = 1$, or $e \leq e_{min}$. To take advantage of this, the IEEE 754 standard records the high bit of C implicitly as the otherwise unused case where E is 0, and keeps track of the rest of C as a shorter bitvector T of size $p - 1$.

The mapping from IEEE 754 bitvectors to reals is given in Equation 4,

$$real(S, E, T) = \begin{cases} NaN & e > e_{max} \wedge c' \neq 0 \\ (-1)^s \times \infty & e > e_{max} \wedge c' = 0 \\ (-1)^s \times 2^e \times ((c' + 2^{p-1}) \times 2^{1-p}) & e_{min} \leq e \leq e_{max} \\ (-1)^s \times 2^{e_{min}} \times (c' \times 2^{1-p}) & e < e_{min} \end{cases} \quad (4)$$

where:

$w = size(E)$

$p = size(T) + 1$

$e_{max} = 2^{w-1} - 1$

$e_{min} = 1 - e_{max}$

$s = uint(S)$

$e = uint(E) - e_{max}$

$c' = uint(T)$.

Let us call this the implicit triple of bitvectors, or "implicit triple" or "IEEE 754 triple" representation of floating point numbers. We also say that the IEEE binary interchange format uses a "packed bitvector" or "implicit packed" or just "packed" representation $B = concat(S, concat(E, T))$. S can always be recovered, and E and T can be recovered as long as w or p is known. Conversion between the packed and implicit represention is expressed in Equations 5 and 6.

$$packf(S, E, T) = \ concat(S, concat(E, T)) \tag{5}$$

$$\begin{aligned} unpackf(B, w, p) = (&extract(w + p - 1, w + p - 1, B), \\ &extract(w + p - 2, p - 1, B), \\ &extract(p - 2, 0, B)) \end{aligned} \tag{6}$$

While packing and unpacking implicit representations is straightforward, converting between the implicit and explicit representations is surprisingly tricky. When going from implicit to explicit, all we need to do is find the correct value of the implicit bit to concatenate with T. Usually this is `0b1`, unless $uint(E) = 0$, in which case it is `0b0`, or $uint(E) = 2^w - 1$, in which case we have to be careful not to accidentally convert infinity into NaN. One way to avoid this is to assume the implicit bit for representations of non-finite reals is always `0b0`, as in Equation 7,

$$explicit(S, E, T) = \begin{cases} (S, E, concat(\texttt{0b0}, T)) & uint(E) = 0 \vee \\ & uint(E) = 2^w - 1 \\ (S, E, concat(\texttt{0b1}, T)) & \text{else} \end{cases} \tag{7}$$

where:

$w = size(E)$.

Going the other way, from the explicit to the implicit representation, is more complicated. First, we need to be able to put an explicitly represented floating point number in canonical form. One technique for doing this is expressed in Equation 8,

$$canonicalize(S, E, C) = \begin{cases} (S, E, C) & e' > e_{max} \\ (S, bv(0, w), C) & c = 0 \\ (S, bv(0, w), C \ll x) & h < z \\ (S, bv(e' - x + e_{max}, w), C \ll x) & \text{else} \end{cases} \tag{8}$$

where:

$$w = size(E)$$

$$e_{max} = 2^{w-1} - 1$$

$$e_{min} = 1 - e_{max}$$

$$e' = max(uint(E) - e_{max}, e_{min})$$

$$c = uint(C)$$

$$z = clz(C)$$

$$h = e' - e_{min}$$

$$x = min(z, h).$$

We find it convenient to distinguish $e' = max(uint(E) - e_{max}, e_{min})$, which is the exponent that will actually be used when computing the real value due to the fourth case in 3 where $e < e_{min}$. Note the use of clz, or "count leading zeros", which counts the number of zeros in the most significant bits of a bitvector before the most significant 1 (or the total number of bits, if all are 0). This is not generally considered a primitive operation in bitvector arithmetic, but it can be implemented efficiently for bitvectors of known size. x is the largest offset we can shift by, to either ensure that $C[p-1] = 1$ or that $e' = e_{min}$. Note also that for all floating point numbers that do not represent finite real numbers, i.e. infinities and NaNs, canonicalization is a no-op, as we need to be careful not to convert infinities to NaNs or vice versa and we have no mapping to real numbers to inform any changes to our representations of NaNs.

With the ability to canonicalize our explicit representation, we can now convert it to the implicit representation used by IEEE 754. Given a canonical explicit representation, we can usually just chop off the most significant (implicit) bit of C to produce the appropriate T. There is still one problematic edge case, when we have a NaN where only the most significant bit of C is 1, as simply chopping that bit off will yield an infinity. To avoid this, we preserve the 1 in the most significant bit of T; this loses some information, but that is unavoidable anyway since there are more explicit NaNs due to the extra bit in C.

Fortunately the conversion doesn't lose any information about represention of real numbers. This is the reason for the fourth case in 3 where $e < e_{min}$ and the max in $e' = max(uint(E) - e_{max}, e_{min})$. We could just as well allow $e = -e_{max}$ in the explicit representation and represent more real numbers closer to zero, but this would cause another edge case in the conversion because those numbers would not be representable with the implicit representation. The explicit to implicit conversion is given in Equation 9,

$$implicit(S,E,C) = \begin{cases} (S_c, E_c, bv(2^{p-2}, p-1)) & uint(E_c) = 2^w - 1 \wedge \\ & uint(C_c) \neq 0 \wedge \\ & uint(T) = 0 \\ (S_c, E_c, T) & \text{else} \end{cases} \tag{9}$$

where:

$(S_c, E_c, C_c) = canonicalize(S, E, C)$

$w = size(E_c)$

$p = size(C_c)$

$T = extract(p-2, 0, C_c)$.

Often when discussing floating point representations, much ado is made about "subnormal" or "denormalized" numbers. We have neglected to discuss them at all, because in our formalization they can be treated the same as any other number. A subnormal number is simply a floating point number with minimum effective exponent $e' = e_{min}$ and a 0 in the high (implicit) bit of C. They are the only numbers with $C[p-1] = 0$ in their canonical explicit representation, and are distinguished in the implicit representation as the only numbers with $e < emin$ (other than the zeros, unless those are considered subnormal). Subnormal numbers are important because they close the gap between the smallest number that can be represented with $C[p-1] = 1$ and 0 in a uniform way, as shown in **??**. Their existence explains the strange contortions we go through defining the relationship between E, e, and e'; once we accept those contortions, their real value at least is easy to specify.

It should be noted that most hardware implementations use the packed bitvector representation; i.e. on a typical implementation using 32-bit single-precision floats, $\frac{4}{3}$ would be represented as

`0b00111111101010101010101010101011`,

which would unpack (assuming $w = 8$, $p = 24$) to the implicit triple

(`0b0`, `0b01111111`, `0b01010101010101010101011`)

or the explicit triple

(`0b0`, `0b01111111`, `0b10101010101010101010101011`).

The floating point theory defined as part of the SMT-LIB standard specifies floating point literals in a form equivalent to the implicit triple representation, i.e. the term of sort `(_ FloatingPoint 8 24)`, where 8 and 24 are w and p respectively, that represents the real number closest to $\frac{4}{3}$, is written

`(fp #b0 #b01111111 #b01010101010101010101011)`.

We should also comment briefly on the possible sizes of bitvectors that can be used in these representations. We require that $size(S) = 1$, $size(E) = w \geq 2$, and $size(C) = size(T) + 1 = p \geq 2$. Since a bitvector has to have at least one bit in our notation, the restriction on p is obvious. The restriction on w arises because a single bit of exponent is insufficient to encode all of the cases needed to describe the implicit bit. If $w = 1$, we have $e_{max} = 2^{w-1} - 1 = 0$ and $e_{min} = 1 - e_{max} = 1$, which means that (perhaps surprisingly) $e_{min} > e_{max}$ and we can never have the case of fininte, non-subnormal where $e_{min} \leq e \leq e_{max}$. Because of this, the explicit representation will be able to represent some real numbers, with representations were $C[p-1] = 1$, that cannot be represented in an implicit representation using the same w and p.

4 Reals as Bitvectors as Ordinals

Floating point numbers represent real numbers, but since they do so with bitvectors, they have some properties that bear more resemblance to integers. For example, it might be more important to know how close two real values r_1 and r_2 are, not in terms of their real difference $r_2 - r_1$, but in terms of the number of representable floating point numbers that exist between them. This quantity, typically referred to as as units in the last place, or "ulps," is entirely dependent on the properties of the floating point representation in question and independent of the properties of real numbers.

To compute ulps, it can be helpful to think of floating point numbers as ordinals. By "ordinals," we mean integers that impose a total ordering on all real values representable with a given floating point representation, not true ordinal numbers in the mathematical sense. A similar notion is presented in the IEEE 754 standard's totalOrder predicate. The floating point numbers of a given representation with finite w and p form a finite set, so a total ordering is possible, and indeed that is what totalOrder provides. However that ordering has some properties that might be surprising: all of the NaNs are ordered, which is necessary for a total ordering but perhaps not helpful, and more problematically -0.0 is less than +0.0, even though both have the same real value.

Several different orderings are possible: we choose one that is hopefully the least surprising for implementers, and discuss how it compares to the totalOrder predicate and other alternatives. We implement this ordering with a function $ord(F)$, which takes a floating point number F in some known representation (such as a packed bitvector or implicit triple with known w and p) and produces a positive or negative integer. We would like $ord(F)$ to have the following properties:

1. If $real(F) = 0$, $ord(F) = 0$. I.e. all floating point numbers that represent a real value of zero (including ones that have a negative sign) have an order of 0.

2. If $real(F_1) = real(F_2)$, then $ord(F_1) = ord(F_2)$. I.e. floating point numbers that represent the same real value have the same order.

3. If $real(F_1) < real(F_2)$, then $ord(F_1) < ord(F_2)$. I.e. floating point numbers are ordered by their real value.

4. If $real(F_1) = -\infty$ and $real(F_2) = \infty$, then $ord(F_2) - ord(F_1)$ is equal to the number of distinct real values the representation can represent (including the two infinities, but not including any NaNs) minus one. I.e. the ordinals are tightly packed: there is no positive i such that there is no F that has $ord(F) = i$ and yet there is some F' that has $ord(F') > i$, or analogously for negative i.

5. If $real(F)$ is NaN, then $ord(F)$ is undefined. In general, the order of a floating point number that represents NaN cannot be relied upon to be meaningful, so the safest thing to do is refuse to assign it meaning. An implementation might want to do something in this case like signal an error, or extend the notion of an ordinal to include NaN.

We can implement $ord(F)$ using arithmetic. The computation for implicit triples is given in Equation 10,

$$ord(S, E, T) = \begin{cases} undefined & u > u_{max} \\ (-1)^s \times u & u \leq u_{max} \end{cases} \tag{10}$$

where:

$w = size(E)$

$p = size(T) + 1$

$u_{max} = (2^w - 1) \times 2^{p-1}$

$s = uint(S)$

$u = (uint(E) \times 2^{p-1}) + uint(T)$.

The new quatities u and u_{max} represent the absolute value of the order and the maximum defined order, or the order of ∞, respectively. Given an implicit triple, u can be computed easily by taking the concatenation of E and T and treating it as an unsigned integer: $u = (uint(E) \times 2^{p-1}) + uint(T) = uint(concat(E, T))$.

Given some ordinal i (and w and p such that $|i| \leq u_{max} = 2^w - 1) \times 2^{p-1}$), we can convert back to an implicit triple according to Equation 11,

$$float(i, w, p) = \begin{cases} (\texttt{0b0}, E, T) & i \geq 0 \\ (\texttt{0b1}, E, T) & i < 0 \end{cases} \tag{11}$$

where:

$u = |i|$

$U = bv(u, w + p - 1)$

$E = extract(w + p - 2, p - 1, U)$

$T = extract(p - 2, 0, U)$.

Because of the way the implicit bit is handled, ordinals are extremely straightforward to work with for the implicit representation. With the implicit bit, there is no overlap of representable real values between different exponents; the subnormals naturally occupy the space closest to 0, as they have the smallest exponent; and the two infinities occupy the points farthest away, even conveniently having $uint(T) = 0$. Conversion formulas are even simpler for the packed representation, as E and T are effectively already concatenated together. These are given in Equations 12 and 13. Again, it is assumed that $|i| \leq u_{max} = 2^w - 1) \times 2^{p-1}$ for the appropriate w and p; larger or smaller values of i are outside the set of ordinals for the floating point representation.

The conversion from packed representation to ordinal is

$$ord(B, w, p) = \begin{cases} undefined & u > u_{max} \\ (-1)^s \times u & u \leq u_{max} \end{cases} \tag{12}$$

where:

$S = extract(w + p - 1, w + p - 1, B)$

$ET = extract(w + p - 2, 0, B)$

$u_{max} = (2^w - 1) \times 2^{p-1}$

$s = uint(S)$

$u = uint(ET)$.

Note that w and p are only needed to check that the real number the ordinal represents isn't NaN. Similarly, the conversion back to a packed representation is

$$float(i, w, p) = \begin{cases} concat(\texttt{0b0}, ET) & i \geq 0 \\ concat(\texttt{0b1}, ET) & i < 0 \end{cases} \tag{13}$$

where:

$u = |i|$

$ET = bv(u, w + p - 1)$.

Here w and p are only needed to figure out the size of the packed bitvector. The ordinal representation is dependent only on their sum, other than the very slight difference in the largest allowed magnitude u_{max}. Moving bits between E and T changes the spacing of real numbers represented, but not their relative ordering.

Ordering is much harder to compute for the explicit representation. Because of the inherent redundancy, it is difficult to write down a simple arithmetic or bitvector expression to convert an explicit triple into an ordinal. Any such algorithm will have to effectively canonicalize the representation, so we might as well just convert it to the implicit representation and take the ordinal of that. We can think of the implicit representation as a way of assigning the canonical floating point representations to reals in a convenient order.

Besides the totalOrder predicate discussed above, our notion of ordinals is closely related to the IEEE 754 standard's nextUp and nextDown operations. These operations produce the "next" immediately smaller or larger number in a floation point representation, i.e. the number with the next greater or smaller ordinal. A number's ordinal is the number of times one must call nextUp (or negative the number of times one must call nextDown, for negative numbers), starting from zero, to reach that number.

We can implement nextUp and nextDown in terms of operations on ordinals, as given by Equations 14 and 15 for implicit triples:

$$nextUp(S,E,T) = \begin{cases} (S,E,T) & i \text{ is } undefined \vee i = u_{max} \\ (bv(1,1), bv(0,w), bv(0,p-1)) & i = -1 \\ float(i+1,w,p) & i < u_{max} \wedge i \neq -1 \end{cases} \tag{14}$$

$$nextDown(S,E,T) = \begin{cases} (S,E,T) & i \text{ is } undefined \vee i = -u_{max} \\ float(i-1,w,p) & i > -u_{max} \end{cases} \tag{15}$$

where:

$w = size(E)$

$p = size(T) + 1$

$u_{max} = (2^w - 1) \times 2^{p-1}$

$i = ord(S,E,T)$.

The extra case in nextUp is required to deal with the sign of zero. The IEEE 754 standard specifies that nextUp should produce negative zero; this is natural given the behavior of rounding small negative values to negative zero. Since our ordinals do not model the sign of zero, we must handle this case separately.

4.1 Ulps: units in the last place

- ulps as ordinal difference
- special cases

  zeros

  infinities

  other pitfalls?
- posix nextafter (test with z3?)
- table of useful constants (1, tiny, umax, etc.)

5 From reals to bitvectors

So far, our discussion of the relationship between real numbers and floating point numbers has been limited to defining which real number a given floating point number represents. This makes it easy to convert from floats to reals, at least on paper: the definition tells us exactly what to calculate. Going the other way, finding the floating point number closest to a given real number, is more challenging. Rather than simple arithmetic, we will have to perform some kind of quantization or search.

5.1 Searching the ordinals

Fortunately, treating floating point numbers as ordinals allows us to search them efficiently. As long as we can compare a real number to the calculated real values of floating point numbers, we can conduct a binary search over the space of ordinals to find two adjacent ordinals whose real values bracket it. This procedure is implemented by the Python code in Figure 5.1.

The first argument `R` is the real number R we are searching for, and the second and third arguments `w` and `p` are the typical exponent size and precision w and p used to define a format. The function returns a tuple of two integers, representing the two ordinals i and j such that $j \geq i$, $j - i \leq 1$, and $real(float(i, w, p)) \leq R \leq real(float(j, w, p))$.

Integers in python are unbounded, so we can assume that all integer arithmetic is exact without having to worry about overflow. The `//` operator is integer "floor" division, rounded down to the nearest integer. The `float` and `real` functions are as defined in Equations 11 and 4. `float` returns a tuple of bitvectors, while `real` returns some representation of a real number that we can compare with our input.

The correctness of the algorithm is easy to sketch out. First, note we have the loop invariant that $real(float(\texttt{below}, w, p)) < R < real(float(\texttt{above}, w, p))$. This is initially true because `above` and `below` are initialized to $\pm u_{max}$ such that $real(float(\pm u_{max}, w, p)) = \pm\infty$. R is a real number, which means that it

Figure 1: Binary search for ordinals bracketing a real number R, implemented in Python

```
def binsearch_nearest_ordinals(R, w, p):
  umax = ((2**w) - 1) * (2 ** (p-1))
  below = -umax
  above = umax
  while above - below > 1:
    between = below + ((above - below) // 2)
    S, E, T = float(between, w, p)
    guess = real(S, E, T)

    if R < guess:
      above = between
    elif guess < R:
      below = between
    else:
      return between, between

  return below, above
```

is strictly not infinite; extending the algorithm to deal with the two infinities is not difficult (just return (u_{max}, u_{max}) for ∞ or $(-u_{max}, -u_{max})$ for $-\infty$).

To show that the loop invariant holds after executing the loop body, there are three cases to consider. The `guess` is the real value of the floating point number with ordinal `between`. If $R <$ `guess`, then we assign `above` = `between`, and the invariant holds. If `guess` $< R$, then we assign `below` = `between`, and the invariant holds. Otherwise, we will break out of the loop and return early.

Because of the loop condition, we know that `below` $<$ `between` $<$ `above`. This implies that the loop will terminate: on each iteration, we either return directly or set one of `above` or `below` to the value of `between`. Since `above` − `below` is finite, and we reduce it each time, it will eventually not be greater than 1 and we will break out of the loop.

Our desired postcondition, mentioned above, is that the fuction returns a tuple of ordinals (i, j) such that $i \leq j$, $j - i \leq 1$, and $real(float(i, w, p)) \leq R \leq real(float(j, w, p))$. There are two return statements where we need to check this. If we return inside the loop, we can see that $i = j =$ `between`, satisfying the first two parts of the postcondition, and also by negation of the conditionals that $R =$ `guess`, satisfying the last part. If we return outside the loop, then the first part must be true because the loop condition held when we calculated `between`, so `below` $<$ `between` $<$ `above`, and assigning `above` = `between` or `below` = `between` will not change the relative order. The second part follows from the negation of the loop condition, and the third follows from the loop invariant because we return ($i =$ `below`, $j =$ `above`).

The complexity of the algorithm is harder to define. Our termination argument is very weak: since it is a binary search, we are actually reducing the space of ordinals to search by half with each iteration, so the number of iterations is $O(\log(u_{max}))$, which is $O(w + p)$, or one iteration for each bit in the representation. Each iteration performs one or two comparisons between real numbers. In theory, all of the comparisons are computable (as one of those real numbers is always rational, since it is represented by some floating point number) but they may be costly if the other number is nontrivial to reason about, such as π.

5.2 Correct IEEE 754 rounding

Binary search will find two floating point numbers that bracket a given real number, but unless they happen to be the same (which implies that the real number is exactly representable) it will not tell us which of the two is closer. Even the definition of "closer" is not immediately obvious. The IEEE 754 standard specifies 5 different rounding modes to clarify.

Three rounding modes are very easy to specify: roundTowardsPositive always rounds up to the larger number; roundTowardsNegative always rounds down to the smaller number; and roundTowardsZero always rounds "down" to the number with smaller absolute value. As with all IEEE 754 rounding modes, rounding a number less than zero up to zero produces negative zero instead.

The other two modes, roundTiesToEven and roundTiesToAway, try to round to the bracketing floating point number that is nearer in value. They differ only in cases where a real number lies exactly between two floating point numbers: as one might expect, roundTiesToEven rounds to whichever floating point value has $T[0] = 0$, while roundTiesToAway rounds to whicever has greater absolute value. Since there is no way to be nearer to infinity than to any finite value (without actually being infinity), these modes use a cutoff value that is the real equivalent of one half ulp greater than the largest representable finite value as the halfway point instead; this value can be calculated as for a format with given w and p as $\pm 2^{e_{max}} \times (2 - \frac{2^{1-p}}{2})$, where as usual $e_{max} = 2^{w-1} - 1$.

Implementing correct rounding given bracketing ordinals or floating point numbers is not particularly challenging, though roundTiesToEven and roundTiesToAway do require some ability to compare differences between real numbers. We will not include any code here because it is long and relatively uninteresting, but a Python implementation can be found in Titanic's core [TODO: [**?**]].

Another way of thinking about rounding is as a set of "rounding envelopes" that cover the real line, including the affine extensions to $-\infty$ and ∞. Each rounding envelope corresponds to some floating point number, and is an interval that exactly describes the set real numbers that will round to that number. For a given format, the envelopes for all of that format's floating point numbers are disjoint (i.e. rounding is a deterministic function) and together they cover all real numbers and infinities (i.e. that function is surjective).

5.3 Floating point arithmetic

The IEEE 754 standard does not specify in any way how arithmetic should be performed. The operations required by the format include basic arithmetic, specifically addition, subtraction, multiplication, division, and square root, which must be "correctly rounded," which is to say they should provide the same answer as rounding the (real) result of applying the appropriate (real) arithmetic operator to the real values of the operands.

For performance reasons, most implementations of IEEE 754 floating point arithmetic implement these operations as circuits (if they're in hardware) or by using low-level bitvector operations. These implementations are complex; describing them in detail is beyond the scope of this document. However, there is a much simpler way to implement correctly-rounded floating point arithmetic, at least on paper. Just use real numbers.

Converting from floating point to reals is easy: just use the formula from the definition, such as Equation 4 for implicit triples. Converting from reals back to floating point is also relatively easy: use the algorithm from Figure 5.1, then round. Since we know how to perform real arithmetic, we can specify the outcome of any floating point arithmetic operation by taking the (floating point) inputs, converting them to reals, performing the (real) arithmetic operation, and finally converting the result back to floating point. If we have a library of programming environment that knows how to do arithmetic on real numbers in addition to comparing them, then this specification is also an implementation of IEEE 754 floating point—although likely a very slow one.

6 Conclusion

Often we like to think of floating point arithmetic as a discrete problem, specified in terms of bits. But while the inputs and outputs are definitely bits, the relationship between them is really specified in terms of real arithmetic. This becomes particularly apparent with less trivial operations, such as powers, logarithms, and trig functions: discrete implementations of the functions are very complex (and not always accurate!) while the specification in terms of rounded real arithmetic is no different than for addition or subtraction.

From a mathematical standpoint, IEEE 754 floating point is little more than a set of rules for quantizing real numbers at specific points during a calculation. There are a few special cases, such as what to do with negative zero, infinities, and expressions like $\infty - \infty$ and $\frac{1}{0}$, but these are not specific to the discrete nature of a representation based on bits and can be handled by a theory that extends real arithmetic, such as in [TODO: [**?**]].

Combined with such a theory, this document should give a complete and formal specification of the output of any expressible floating point computation in any IEEE-754 like binary format. This specification is "infinite precision," as it can be adapted to any output format, and error due to rounding of intermediate results can be avoided simply by removing quantization operations. Beyond the

outcomes of computations, it can be used to define other quantities as well, such as real valued error or ulps difference due to rounding of intermediate results.

6.1 Titanic: guaranteed to float correctly

This document is just a paper specification: it can't do any computation on its own. To make up for that weakness, we provide a concrete implementation of the specification in Python. We call the implementation Titanic (with appropriate irony), as it is a floating point library "guaranteed to float correctly."

Most floating point libraries are designed for performing numerical computations; to that end, they are optimized for performance. Titanic, in contrast, is designed for checking that numerical computations are well behaved; to that end, it is optimized for simplicity and obvious correctness. Titanic uses SymPy [TODO: cite] to perform real arithmetic; the core theory relating floating point numbers and real numbers is implemented exactly as described here. Titanic is available online at [TODO: source and stuff] titanic.uwplse.org.

References

[1] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[2] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, (New York, NY, USA), pp. 135–152, ACM, 2013.