To Infinity and Beyond

Richard J. Fateman

Electrical Engineering and Computer Sciences Dept.
University of California, Berkeley, 94720-1776, USA

April 13, 2016

Abstract

Computer algebra systems (CAS) are sometimes expected to compute with infinities symbolically. Since operations on such objects do not always conform to a sufficient and consistent set of rules for ordinary arithmetic, some compromises are necessary. We discuss a few options and implementations.

1 Introduction

How much computation can be supported by a computer algebra system (CAS) using objects $\pm\infty$ or "indefinite"? These objects have certain aspects of numerical constants, but simply fail to satisfy some essential properties of conventional "real numbers". In particular they do not satisfy mathematical axioms for elements of algebraic fields. It is hardly ever a good idea to expect much useful information by continuing to perform at any length, arithmetic operations on these objects unless they are mapped into real numbers (as, say $1/\infty$ becomes 0). As this illustrates, on occasion a few operations are necessary to run a computation to an unexceptional conclusion. What operations are sensible to allow in these circumstances? Here we explore a few options that can be implemented. A paper by Beeson and Wiedijk [**?**] provides some formalization of semantics and comparison with Mathematica's routines, and reinforces our evidence that there is a synergy with interval computation.

2 Why Allow This At All?

CAS may use terms such as `inf, infinity, Infinity, DirectedInfinity, ComplexInfinity` to denote some version of the concept of an unbounded real or perhaps complex number. Since we are typesetting this paper, let us denote the object under scrutiny, infinity, as ∞. If humans use terms like ∞ in mathematical discourse, then the domain of applicability of a CAS (which includes "all" mathematics, at least in principle) must deal with this. In practice, a CAS, at best, includes as much as has been programmed, and the rules for ∞ do not fall out automatically as a result of the usual axiomatic definitions.

We routinely expect that we can write a command to compute an integral from 0 to ∞, or that we may receive a computed result from integration or limit calculations that is ∞. The notation also ∞ appears in a well-defined context as an unbounded upper limit (or its negation as a lower bound) in interval notation: Let [a,b] represent all (real) numbers x such that $a \le x \le b$. In this context the notation $[0,\infty]$ conventionally means all real x such that $0 \le x < \infty$ (not $x = \infty$, which turns out to be problematical).

In addition to integration and limit contexts for ∞ it might appear as results of summations or asymptotic calculations, or the values of functions at singularities. It also might appear as a marker in other uses, as indicating that the "precision" of an exact rational number is infinite, or even that a loop index should proceed to higher and higher values without limit.

Consequently, we simply cannot shy away from ∞ situations without sacrificing part of the "expected" domain of discourse for these systems.

3 What's the Problem?

The symbol ∞ does not obey the ordinary rules of arithmetic.

Example: we routinely solve $x \times y = 1$ for y, with an answer $y = 1/x$. Yet that cannot be right if we allow $x = \infty$. If ∞ has any meaning, it is that $1/\infty$ should be 0. Checking by back-substitution into $x \times y$, we come up with the apparently false statement that $\infty \times 0 = 1$.

In general, if we mix ∞ elements into our algebraic domain or even use programming assignments such as `x:=`∞, then we open our system to several sources of contradictions resulting from the failure of `x` to satisfy rules for the real numbers that are implicitly or explicitly encoded in the procedures of a CAS, rules that fail for the *extended* real numbers[1] .

4 CAS Can Overcome Some Problems

Let us distinguish CAS "computational mathematics" from conventional mathematics where a stricture "division by zero is forbidden" is perfectly acceptable. From a constructive perspective, it fails to provide guidance as to how a programmed procedure might respond usefully *when a user in fact types 1/0.* Simply put: the programmer must write a procedure that will do *something.* It might return a value or it might just halt with an error message. There is a distinct possibility in a symbolic system, one that is not available in a numerical one, which is that the symbolic system may just keep "1/0" symbolically. This may work but if arithmetic expressions on numbers are not evaluated to numbers, There is the somewhat unappetizing prospect that an expected simple numerical result will be an expression with embedded "1/0" and "0/0" forms.

Let's elaborate somewhat on the the distinction between numeric and symbolic. The viewpoint of typical (numerical) computational scientists who use floating-point numbers effectively in applications use representations for the numbers within strict boundaries (e.g. 64-bit floats). This delimits what is possible to compute. To some extent the availability of bit patterns in IEEE 754 floating-point for ∞, -∞, and 0/0 take a step toward enabling computing with expressions that are not "real numbers" but not the full generality of symbolic expressions. Computationally, the floating-point infinity might merely be a number that has overflowed the exponent range and might thus be something quite finite in a wider floating-point format. Regardless, the practical support for computing with such objects is burdened by inadequate or entirely missing language or software resources and clumsy, slow, or even missing hardware features. An important aspect of the IEEE 754 standard in regard to computing with ∞ is that it provided a common basis for addressing some of the issues, even if they were not all laid to rest. And even if some subsequent implementations fall short.

We can solve more of the problems by implementing a more robust system, even if consequently slower. We are willing to haul around extra mechanisms during computation – as we do for so much else in a CAS – a system in which symbols as well as numbers are first-class objects. For example, we know that it might be valid to reduce ∞x to ∞, if x, though strictly speaking "a symbol of unknown value" has the property that it can only be a positive real number. Alternatively, -∞ if x is unknown but assumed negative. Finally if nothing is known about x and it might even be zero, the result should be indefinite. We will use the notation 0/0 as a brief notation for indefinite, regardless of the originating "failure to compute".

It seems that a numerical program with only 64-bits of storage, numbers not symbols, must choose one of those three results, $infty$, $-\infty$, or 0/0 (the latter encoded as a NotANumber (NaN) of which there are different encodings possible).

[1] |`https://en.wikipedia.org/wiki/Extended_real_number_line`

One option for the CAS is to *keep the expression* ∞*x unresolved until or unless x is effectively resolved to non-zero.

This is slightly reminscent of the situation that comes up when $(9)^(1/2)$ is replaced by 3, but the CAS may just encode something like RootOf($x^2 - 9$,x), indicating "one of, but we are not saying which one of" the roots.

There does seem to be a fairly irremediable hazard in that today's CAS will ordinarily simplify $0 \times x$ to 0. This implies that x must never be $\pm\infty$ or 0/0.

How can we attempt to fix this? One way is to refuse to allow x to attain those forbidden values by substitution, binding, or any naming. This leads to considerable complexity: for example, x might be an expression such as $\sin(y)$ in which case the forbidden zone would would be $y = n\pi$ for integer n. Such a line of reasoning sufficient to keep track of assumptions along the path of computation, would convert a CAS into a Truth Maintenance System. A TMS would be appropriate for a system intended for use in proving mathematical theorems, but (for those who have considered it) the associated costs have been discouraging. The inclusion of such facilities in widely-used CAS would incur costs for computing time and storage, but another challenge is devising procedures to adequately simplify the side-condition expressions is daunting.

5 One Approach: Embroidering ∞ and Friends with Direction and Index

With some forethought a CAS designer can build more information into ∞. In particular, each time an ∞ is generated, it can be given an index, `infinity[c1]` or perhaps `infinity(c1)`. Here `c1` is a count, which is incremented each time we utter a new infinity. Then we can compute the difference of two infinities that are "the same" by virtue of holding the same index, and they can cancel out to zero. A further elaboration, already adopted by the Mathematica notation `DirectedInfinity[]` is to also associate a direction. For instance `infinity(1,c1)` Holds the 1 as direction in the complex plane, hence this is real, positive. -∞ becomes `infinity(-1,c2)`. It is plausible to modify the display to be unfaithful to the internal representation and completely omit the mention of the index, and for that matter, remove the direction as well, with the two directions for real infinity displayed as ∞ and $-\infty$ respectively, as `infinity(%i)` could be displayed as i ∞. For other directions, Mathematica computes a unit normal so that the direction $1 + i$ is simplified and displayed as $(1 + i)/\sqrt{2}\infty$.

In Maxima, if n is a numeric constant and x is a symbol, then ∞`*x*n` can be converted to something like `x*infinity(signum(n),c3)`.

This can be tied in to some existing uses of `infinity` using Maxima's syntax extension:

```
infcounter:0;
nofix(infinity);
infinity:= (infcounter:infcounter+1, myinfinity(1,infcounter));
```

There is no widely-accepted standard notation for the result of computing 0/0 or other undefined operations, but there is a collection of possibilities in various contexts[2]. These include NaN (Not a [legal floating-point] number), NaI (Not an Interval), Unknown, indeterminate, undefined, null, empty interval, (an interval enclosing no numbers at all), bottom ($\perp$). There may be subtle variations. For example, indefinite or indeterminate might be a number known as real and finite, but its value is not specified, whereas $\perp$ probably signifies that there is no possible number. Which might be the same as an empty interval. and thus the concepts should not be separately notated. Mathematica says 0/0 is indeterminate. It says limit(sin(1/x),x,0) is an interval [-1,1]. and that limit(x*sin(x),x,inf) is also an interval [-oo,oo]; Maxima says this first limit is indefinite ("ind"). Maxima says the second limit is undefined ("und").

[2]See Beeson [1] for even more.

(traditionally an interval is used to denote some particular number, one that is unspecified but whose value lies between the upper and lower bounds...) this usage in Mathematica returns a range or subset of the real line, from -1 to 1. In this second limit example, the expression assumes *any* real number an infinite number of times if one considers approaching the limit as a process. Interval arithmetic advocates refer to the variant used in these cases (but not some others) by Mathematica as containment sets or cset intervals. The forthcoming IEEE 1788 standard for interval arithmetic declines to endorse this cset interpretation.) Csets are nicely discussed in this paper, which among other benefits describes the trade-offs between design objectives: `http://www.cas.mcmaster.ca/~isl/Publications/IntvlArithCsets.pdf`.

Adopting the above design, as well as other proposals to follow, requires a significant programming change in that every product being simplified must be examined to see if it (a) includes an `infinity()`, (b) includes zero, or (c) includes both. There is the uncomfortable and irresolvable difficult that might emerge, if through assignment some subexpression might be infinity later.

We have assumed the function `signum(n)`, which computes a unit-normal vector in the complex plane, will be used only on numbers, but conceivable it could be carried around symbolically in case `n` is later is given a value.

In the same vein we can also consider `infinity(0,c6)` to be a special case meaning 0/0 or indeterminate. An alternative interpretion might be "non-directional complex infinity", in which case we are left without 0/0. We could solve that with some other symbol. Or just leave that alone unchanged.

Following the implications of this development another consideration turns up: the encoding of a signed zero. It is useful to have $1/(1/x)) = x$ for all x *even including* $x = 0$ and $x = \pm\infty$. That is, $1/(-\infty)$ should be -0, and $1/(-0)$ should be $-\infty$. With a signed zero, we can do this. Otherwise we end up identifying only one value for $\pm\infty$, a projective model, mathematically speaking.

An alternative to the signed zero might be introducing infinitesimals: adding `tiny(1,c3)` for $1/\infty$ and `tiny(-1,c4)` for $-1/\infty$. We considered this but rejected it as too complicated for explanation or for implementation.

There is a history of ad hoc implementation (in Macsyma, predecessor to Maxima) of some of these ideas dating back to the 1970s, in order to facilitate programs for limit calculation and definite integration. Currently Maxima has `inf, minf,` as well as `zeroa, zerob` (respectively "above zero" and "below zero") which, according to the documentation, can be used in expressions, but whose special attributes are entirely ignored unless one encloses the expression containing them in `limit()`. This invokes a special simplifier. Often these special symbols are simplified away *before their properties are examined*, so for example `limit(inf-inf)` is reduced to `limit(0)` by replacing `inf-inf` by 0 before the special simplifier is used! The result is 0. This substantially constrains its usefulness. In tracing through the Maxima's limit() simplifier, numerous subexpressions involving floating point numbers related to 1.0e-8 suggest that some heuristics are involved.

Handling these items as well as the new ones we propose gets too complicated when they all must be considered as "first-class" arithmetical objects, not just within the special confines of the `limit()` command. The existing system programs must be augmented to make sense of them when possible *in any context in which a number can appear*: as arguments to trigonometric functions, exponents, comparisons, conversion to numeric floating-point functions, bigfloats, etc.

Finding a universally agreed upon result in all cases is likely not all possible given the axiomatic failures, but even a "best efforts" attempt is complicated by coming late to the table – when so much else is already coded, and usually code elaborated[3] to handle other special cases over the years. It is also mostly orthogonal to what is proposed in the next two sections.

The two proposals outlined in the next two sections can be implemented mostly independently of each other, but can also be used, with some benefit, together.

[3]Some would say, encrusted.

6 The Rational Number Alternative

Common Lisp, Maxima, and other CAS as well as some other new programming languages[4]. supportive of scientific computing have a basic data type of "rational number". For specificity we continue to consider Maxima, which already has a complete infrastructure for rational numbers like 1/3. Maxima doesn't quite use it, for integers like 2, preferring instead the Lisp integer 2. But with a modest amount of work we can identify 2 with 2/1, a number with a numerator of 2 and a denominator of 1. We may not even store the denominator, but we could (if the numerator is 0).

For historical reasons Maxima constructs its own rational numbers out of arbitrary-precision integers and list cells. It does not use Common Lisp's built-in rational numbers [5].

We have previously proposed changing the built-in CL rational numbers [2], but to our knowledge this has not been adopted by any implementations. Using Maxima we can program around it, and so we have done so.

The Maxima rational number uses a list structure of three items: a header, numerator integer and denominator integer. For example, the number 1/2 looks like a list of three items `((rat) 1 2)`. Conventionally, the denominator is a positive integer. The header generally is "flagged" as simplified, hence its simplified internal form in Lisp will appear as `(rat simp) 1 2)` would be the simplified form. In principle an expression could have other "flags" added after the `simp`, but in past practice knowing that something is a rational number covered the properties it might have[6].

Unconventionally, and the way we are proposing to use this, is to allow a few formerly forbidden numbers to be represented as we suggested previously [2]. In fact, this earlier paper includes more references and details on design. Essentially, instead of signalling an error related to division by zero, we allow the production of `((rat) 1 0)` which will be our new ∞. Similarly, `((rat) -1 0)` is -∞, and `((rat) 0 0)` or 0/0 is represented. We can also allow one more form, namely `((rat) 0 -1)` to be negative 0.

Note that there is only one positive real infinity, and so we have dispensed with the indexing tags, and the direction. It is still possible to multiply the real infinity by complex numbers, though.

We have written this as a patch that can be added to any recent version of Maxima. It over-writes several pieces of the simplifier to incorporate a few very small code segments to accommodate the new rational numbers[7].

Surprisingly perhaps, the new functionality is provided in part by removing some code, making programs simpler. As indicated above, we eliminate error messages and just produce a new item. This is aided by the convention that `gcd(n,0)` is `n`, and therefore `gcd(0,0)` is 0. 1/2+1/0 is (1*0+1*2)/(2*0), reduced to lowest terms is 1/0. So 1/2 + ∞ is ∞, and this just falls out of the regular computation. This approach requires that we separate arithmetic on rational numbers and integers from addition of "other stuff", and that we don't combine them except with careful rules. Thus `x+1/0` is just that. If, later, `x` is assigned any real value, we can simplify to `1/0`. This requires some extra processing, and while the cost may appear insignificant, checking and rechecking during simplification for 1/0 is necessary only if we've patched the system to allow it.

We can also produce -0 by typing it in, or computing it as for example 0/-5. While it seems we could with small additional effort represent n/d as -n/-d, a numerically equal value, we have not found much use for it, and a number of reasons to forbid it. While a zero with an independently set sign seems almost familiar [8],

[4] e.g. Julia

[5] The original host Lisp implementation (PDP-6 MACLISP) did not have built-in rational numbers. Converting Maxima code to use the Common Lisp (CL) numbers has been suggested, but would be a ticklish enterprise with no clear benefit. A CL rational is an atom. Maxima's rational is a list. Code that takes different branches for atoms and lists would find rational numbers on the wrong side. Additionally, using the CL rational numbers would break the scheme described in this paper!

[6] but see below.

[7] Although the changes are small, they involve some of the larger procedures in Maxima, namely `simptimes, simplus, simpexpt`, whose functions can be guessed from their names.

[8] At least for those who have encountered "one's complement" arithmetic computer hardware or studied the IEEE 754 binary floating-point standard.

having an infinitesimal range around other real numbers is less coherent a notion.

We do not claim an implementation that *fully* integrates this model into Maxima: it would be necessary to patch every place (prominently, limit, sum, product and integration computation) where current infinities are used and produced, as well as setting a policy for conversions to and from machine floating-point NaNs and infinities. There is (currently at least) no representation for these special objects in bigfloats, where there is no limit on the exponent value. The `((rat) n d)` structure is widely used, and there may be programs that we have not located that implicitly require the denominator to be strictly positive, causing difficulty with -0. It is quite evident that many programs, user-written or built-in need no changes: In the absence of infinity operands, they will execute the same arithmetic steps as usual. Many procedures will produce the expected infinite or indefinite results as a consequence of following the usual arithmetic routines.

The interface to floating-point computation is a problem area. Conventionally, addition of a rational to a float produces a float[9]. If that rational is 1/0, we must intercede and make the result 1/0, rather than a float (version of) infinity. Only finite floating-point numbers are specified in the Common Lisp standard, and so the floating-point support in Lisp implementations tends to be somewhat brittle. The support for floats underlying Maxima, and on different computers, vary. One possible uniform response would be to refuse to explicitly or implicitly convert to or from any floating-point version of 1/0, or combining types, eliminating the prospect of interacting with IEEE float infinities. Given that numerical subroutine libraries may use these objects, it would be better to make use of them as appropriate.

Compared to the embroidered infinity of the earlier section, this approach gives us just one ∞, 1/0 encoded as `((rat simp) 1 0)`, one -∞ as `((rat simp) -1 0)`, two zeros: -0 as `((rat simp) 0 -1)`, +0 as the integer 0, and indefinite as `((rat simp) 0 0)`. In our implementation there is only one positive infinity, and so 1/0-1/0 must always return the same value (which is 0/0).

It might seem that in some circumstances we could identify objects that should cancel if we notice that the two 1/0 are lists stored in the same memory location (Lisp's EQ predicate tests this). We cannot use this tactic because a computation like $u = 3 + q$, $v = 3 * q$, where $q = \infty$ will return the very same location for the value of u, v and q.

A different implementation immune to that defect, would produce each ∞ as `((rat simp INDEX) 1 0)` where INDEX would be a unique integer incremented after each production of a new ∞, just as our previous proposal. Any operation with such a number would wipe out the index and if the result was again 1/0 or 0/0, it would be tagged again with a different index. This dependency consideration is well-studied in the context of interval calculations, when [-1,1] - [-1,1] is [-2,2] if those two intervals are independent. but 0 when they are "the same". This is not included in the current implementation but could be easily inserted.

It is possible to provide a directed (complex) infinity in this model by, for example, producing $\infty \times (1+i)$. This needs to be handled cautiously because $\infty \times (3+i)$ if "multiplied through" to $3 \times \infty + i \times \infty$ simplifies to $\infty + i \times \infty$, which is a different direction[10].

There are functions that operate on exceptional operands and return ordinary numbers. For instance, `exp(-1/0)` will simplify to 0. The system knows that `(1/0>0)` but to make this work: `(1/0>0.0)` required inserting a check before conversion to floating-point. While this is a small change, it is one that is repeatedly executed in a main loop of the simplifier[11], rather than, as in the gcd situation, one that eliminates a check. Each log, exponential, and trigonometric function must be updated to deal with 1/0 and 0/0. We have illustrated how this would work with the sine() function in our code file. Unfortunately each of many functions must have a similar small patch.

[9] In Common Lisp as well as Maxima

[10] Mathematica commits the alternative error, not allowing $\infty + i\infty$ to be added properly.

[11] For the Lisp source-code reader, it is extra code in function `addk`

7 The Rational Expression CRE Alternative

The Maxima system has a subsystem for manipulation of canonical rational expressions (CRE). In today's vernacular this would be called an object system defining representations and operations. In essence, each CRE object is stored as a ratio of two polynomials p,q with integer coefficients. Each polynomial can be in any number of variables and p/q is expressed in lowest terms by removing any common divisors. In order to make this unique (hence canonical), the order of variables in p, q ($x < y < z$ for example) is fixed; the order is indicated as part of each object.

Internally within Maxima, that is, viewing the data in Lisp, a CRE looks approximately like `((mrat <ordering info>) numerator . denominator)`, where there is an efficient special encoding of the polynomial terms as a list of exponents and coefficients. The user interface to CRE representation consists of two functions, `rat()` and `ratdisrep()`. `Ratdisrep()` is rarely used since the display functionality is able to show the user CRE forms too. The user typically engages the use of CRE form for a calculation by just one use of `rat()`. Since this representation is "contagious", `rat(1)+x` is the same as `rat(x+1)`, and so a long computation can usually be converted entirely to CRE form by changing one input to be CRE form.

We can use this CRE information structure to store our extension to the rationals using trivial polynomials 1, 0, -1. For instance oo is `((mrat) 1 . 0)`. In more detail, the form is `((mrat simp` *`varlist genvars`*`) num . denom)` with a "dotted pair" lisp notation at the end separating numerator and denominator. *Varlist* is in general a (super) set of the variables in the polynomial numerator and denominator, and *genvars* is a list of the generated symbols used internally in the polynomials. The details of the polynomial representation are unimportant except to note that the polynomial 2 is represented simply by the lisp integer 2. (etc).

Almost all the program procedures used for manipulating these forms are much smaller than those altered for the previous section, and are contained in a small set of files for CRE form. The exceptions are some slight changes for providing input and output. If we restricted our introduction of extensions to the CRE package, *there would be no efficiency affect to the general simplifier.*

Along with this alternative comes some assumption that we have thought through somewhat more explicitly. That is, the variables in our polynomials are assumed to be (ultimately) governed by the rules of combination for finite real (or complex) numbers, but the manipulation is formal and algebraic. Thus `rat((x^2-1)/(x+1))` is simplified to `x-1`. One might ask if this is valid if $x = -1$. Or for that matter, for $x = 1/0$. In the algebraic structure (officially an algebraic Field), the answer is yes, it is formally valid. This leads us to a simplification for our extended version of CRE. That is, if we have `rat(1/0)*x` for some symbol x, and we just follow the rules and compute, for the numerator polynomial, `1*x`, the denominator polynomial `0*1`, we compute the formal `gcd(x, 0)=x`. Reducing to lowest terms by dividing numerator and denominator by x, we get 1/0.

Is this justified? What if $x < 0$ or $x = 0$? Indeed, if `rat(1/0)*0` is explicitly presented, we compute 0/0, or indeterminate as the result. This cannot be entirely consistent since ∞*x is ∞; so is ∞*(-x). But `0*x` is 0. One way of interpreting this is to assert that the CRE form *has only one infinity whose sign is unknown.* This "projective" model has its uses, but is different from the "affine" model with two real signed ∞ (and two zeros). This alternative just falls out of the arithmetic, essentially from the gcd routine.

The CRE approach somewhat lessens to need to make all of these items first-class objects *outside* the CRE subsystem. As constants such as `exp(1/3)` are just represented symbolically as a kernel with respect to the CRE form, so is `exp(-1/0)` . Since there is a Maxima command that will try to "numerically evaluate" `ev(..., numer)`, it is plausible that we can construct a route from these new representation to some floats. In fact, for the (many) Lisp systems that support IEEE 754 floating-point arithmetic, including notations and underlying numerical support for float infinities and NaN which is like our indeterminate, the mapping can be reasonably direct for many functionalities.

7.1 What about just using floats?

If IEEE 754 floating-point arithmetic is available, can we just use floats? We could narrow the focus to one precision, say double-float-infinity and see if that can be used for the representation.

There is a philosophical question in the air here. The CAS can represent objects which are not numbers. Why use the number format to encode and constrain what can be done with non-numbers when we have a whole word of symbolic processing at our disposal? On the positive side though, if (and this is a big if) the underlying support system for IEEE 754 exceptional operands is robust, including operations with elementary functions, etc, this might be a shortcut to introducing these concepts. Unfortunately the level of support in Common Lisp implementations runs the gamut from "not allowed at all" to "it's there, including infinities, directed roundings, and NaNs but use at your own risk". This is hardly unique to Common Lisp: most other programming language standards are under-specified in this area as well.

As a compromise, if there were appropriate standardized support for IEEE 754 floats we could choose a precision, say double-float, and have the symbolic processing use the IEEE 754 representation in place of 1/0, 0/0, etc. This would consolidate the otherwise duplicative notions. This would give us two (signed) infinities, two zeros, and a representation (actually many) for indefinite via NaN[12].

We might quibble this way, though. Since the float ∞ may be produced as the result of a floating-point computation that just slightly overflows, heuristically identifying it as the same ∞ in a symbolic sense is questionable. After all, The CAS can represent numbers as bigfloats or exact rationals that far far exceed the overflow threshold of the machine floats, and thus "floating overflow" could be handled by promotion to "bigfloat" implementations in software, rather than ∞.

8 Examples

9 Comparison with other systems

9.1 Maxima's previous system

Since all the special names are just symbols when they occur in expressions, they suffer all the maladies of ignoring their special properties in ordinary contexts. If they can survive the simplification process, then calling limit() will assert additional simplifications, including for example, `limit(1/inf) = 0`.

9.2 Mathematica

Mathematica 10.l0.2.0 has `Null, Undefined, Indeterminate, Infinity, ComplexInfinity, DirectedInfinity[], DirectedInfinity[direction]`.

Examples, not necessarily ones that show the program in the best light,

`h=If[False,x]` assigns the value `Null` to `h`. Since a returned value of `Null` is not printed, this may be disconcerting. However, `1/h` returns `1/Null`.

`Indeterminate+Undefined` is `Indeterminate`, which according to the documentation is for representing "a numerical quantity whose magnitude cannot be determined." Who knew that adding it to `Undefined`, "a quantity with no defined value" would make it numerical.

The test `Infinity==Infinity` returns `True` and yet
`Infinity-Infinity` is not 0 but `Indeterminate`.

[12] Slight complication: there may also be single or extended float infinities etc.

The test `1/(1/Infinity) ==(1/(1/(-Infinity)))` returns as
`ComplexInfinity==ComplexInfinity`, neither true nor false.
`1/0` returns as ComplexInfinity

`-ComplexInfinity` returns as ComplexInfinity.
but
`Limit[1/x,x->0]` is Infinity

`Limit[1/x,x->0,Direction->-1]` is still Infinity but
`Limit[-1/x,x->0,Direction->-1]` is -Infinity, (which is outright wrong).

0*Undefined is Undefined
DirectedInfinity[] is simplified to ComplexInfinity,
"a quantity with infinite magnitude but underdermined complex phase."
Arg[ComplexInfinity] is Interval[-Pi,Pi]

Arg[Infinity] is 0

Sin[Infinity] is Interval[-1,1].

Null-Null is 0. 0*Null is 0, Indeterminate*Null is Indeterminate, 4*Null is 4*Null.

9.3 Maple

What to say here?

10 Availability

Maxima itself can be downloaded as an executable binary from sourceforge, details depending on your host computer. You still have to choose a version of infinity-handling. Loading `http://www.cs.berkeley.edu/~fateman/lisp/i` into Maxima provides the representation of 1/0 and friends as rational numbers. The default setting of `ratinf:true` enables this. Setting `ratinf:false` returns the system to its previous behavior.

For representing infinities as subscripted/indexed items, the file `maxima-interval.lisp` can be loaded. Then `inf()` produces an indexed representation of a real positive infinity. This has appeared to be useful primarily in an implementation of interval arithmetic.

Neither of these schemes has been entirely integrated into Maxima.

11 Acknowledgments

Thanks for discussions on the maxima newsgroup, the IEEE-754 committee and the IEEE-1788 committee. Many thanks for the numerous explorations of related topics, over many years, with Prof. W. Kahan at Berkeley.

12 Tables of Operations

These are quoted with renumbering and minor typographic changes from Fateman[2].

The extended affine rationals participate in the ordering of finite rationals as follows (for rational $x > 0$):

$$-\infty < -x < -0 < 0 = +0 < x < +\infty$$

. But NaN does not participate in that ordering. In particular, all of the relations $y > z$ $y \geq z$, $y \leq z$, and $y < z$ are false if y or z and $y \neq z$ is true if y or z is NaN. $NaN \neq NaN$.

+	+0	−0	1/0	−1/0	0/0	x
+0	+0	+0	1/0	−1/0	0/0	x
−0	+0	−0	1/0	−1/0	0/0	x
1/0	1/0	1/0	1/0	0/0	0/0	1/0
−1/0	−1/0	−1/0	0/0	−1/0	0/0	−1/0
0/0	0/0	0/0	0/0	0/0	0/0	0/0
y	y	y	1/0	−1/0	0/0	$x + y$

Table 1: Affine "+"

Let $s = \text{sign}(x)$ and $t = \text{sign}(y)$.

$*$	+0	−0	1/0	−1/0	0/0	x
+0	+0	−0	0/0	0/0	0/0	$0 * s$
−0	−0	0	0/0	0/0	0/0	$-0 * s$
1/0	0/0	0/0	1/0	−1/0	0/0	$s/0$
−1/0	0/0	0/0	−1/0	1/0	0/0	$-s/0$
0/0	0/0	0/0	0/0	0/0	0/0	0/0
y	$0 * t$	$-0 * t$	$t/0$	$-t/0$	0/0	xy

Table 2: Affine "$*$"

−	+0	−0	1/0	−1/0	0/0	x
	−0	+0	−1/0	1/0	0/0	$-x$

Table 3: Affine Unary "-"

1/	+0	−0	1/0	−1/0	0/0	x
	1/0	−1/0	+0	−0	0/0	$1/x$

Table 4: Affine "1/"

References

[1] Michael Beeson and Freek Wiedijk, "The meaning of infinity in calculus and computer algebra systems," Journal of Symbolic Computation Volume 39 Issue 5, May, 2005 Pages 523-538 `https://www.cs.sjsu.edu/~beeson/Papers/LimitTheory.pdf`

[2] Richard Fateman and Tak W. Yan, Computation with the Extended Rational Numbers and an Application to Interval Arithmetic, 1994 International Symposium on Symbolic and Algebraic Computation, Oxford UK.