To Infinity and Beyond

Richard J. Fateman

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

January 15, 2022

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 the notion of mathematical infinity ($\pm\infty$) and the related notion of "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 in this example, sometimes one or a few operations can be helpful to complete a computation resulting in an unexceptional result. What operations are sensible to allow in these circumstances? Here we explore a few options that can be implemented. These ideas have an important role to play in interval computation and *vice versa.* A paper by Beeson and Wiedijk [1] 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 $a \leq x \leq b$. In this context the notation [0,∞] conventionally means all real x such that $0 \leq x < \infty$ (not $x = \infty$, which turns out to be problematical).

In addition to integration and limit contexts (input and results), ∞ may be used in summations or asymptotic calculations, or to denote 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?

In short, 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 $x = \infty$. Since it is generally agreed that $1/\infty$ should be 0, how does that work? 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. In the latter, a stricture "division by zero is forbidden" is perfectly acceptable. From a constructive algorithmic (CAS) perspective, it fails to provide guidance as to how a programmed procedure might respond usefully in situations that may occur. Indeed, it must determine what should be done *when a user in fact just 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. It is also the case that 1/0 will appear in the midst of a longer computation, so it is not sufficient to just forbid the typing of that expression. 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 generally highly effectively in applications use representations for the useful numbers within strict boundaries (e.g. representable as 64-bit floats). This delimits what is possible to compute. To some extent the availability of bit patterns in IEEE 754 [2] floating-point for ∞, -∞, and 0/0 take a step toward enabling computing with expressions that are not within the boundaries of the representable numbers, yet do not approach the full generality of symbolic expressions[3]. . 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 provides a common basis for addressing some of the issues, even if all aspects are not laid to rest. Even so, some subsequent "implementations" fall short of providing access to these features.

We can solve more of the problems by implementing a more robust system, even if it is consequently slower. From a CAS perspective we are willing to haul around extra mechanisms during computation

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

[2] The new, 2019 standard is given at
`|https://ieeexplore.ieee.org/document/8766229|`. There is an extensive literature on the earlier, 2008, standard.

[3] Gustafson argues that with 4 bits we can formulate SORNs, Sets Of Real Numbers, in which case 0/0 is "everything". See http://www.johngustafson.net/presentations/Multicore2016-JLG.pdf

involving infinity — as we do for so much else in a CAS — in which symbols as well as numbers are first-class objects. For example, we know that it might be valid sometimes to reduce $\infty \times x$ to ∞, if x, though strictly speaking "a symbol of unknown value" is partially determined. For instance, a CAS typically may have a feature that allows a user to declare that x can only assume values that are positive reals. Alternatively, we can return -∞ if x is unknown but assumed negative. Finally if nothing is known about x and it might even be zero or imaginary, 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 numbers fitting in only 64-bits of storage must choose one of those three results, $infty$, $-\infty$, or 0/0 (the latter encoded as some NotANumber (NaN) of which there are different encodings possible). One option for the CAS is to *keep the expression* ∞*x unresolved until or unless x is effectively resolved to a particular value.

This is slightly reminscent of the situation that comes up when $9^{1/2}$ might be replaced by 3, but the CAS may be more cautious since -3 is possible in some contexts. It 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, in the interests of getting work done, will 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 ∞. It is more difficult to retrofit this idea into an existing system, but here's a possible design. Each time an ∞ is generated, either by typing in `infinity` or as the result of a computation, it can be encoded with a unique index, `infinity[c1]`. Here `c1` is a count to make it unique. The count is incremented each time we utter a new infinity[4]. 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 "positive real" in the complex plane. A negative value -∞ 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 usually remove the direction as well, with the two directions for real infinity positive and negative displayed as ∞ and $-\infty$ respectively. An `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 complex constant and x is a symbol, then ∞`*x*n` can be converted to something like `x*infinity[signum(n),c3]`. The `signum` function returns a unit normal.

One way to harness this idea is easy to implement in Maxima. We can redefine `infinity` using Maxima's syntax extension.

```
infcounter:0;
```

[4]there are subtleties in saving such items: two separate runs must have non-intersecting index sets for infinities if the data from both is restored to a single process.

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

Then a measure of back-compatibility with old code is provided: a prior utterance of `infinity` is converted to tt myinfinity(1,k) for some integer `k`. And some encoding of `myinfinity` as a function, an array marker, or perhaps an extended rational number 1/0. This does not solve all compatibility issues since existing code might (say) test to see if the limit of an integration is `infinity` by some comparison of symbol names. It needs to be somewhat wiser.

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[5]. 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. If we first agreed on the minimal set of concepts we could weed out redundant notations. 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").

(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 approaching the limit – if one extends the limit concept. Interval arithmetic advocates refer to the variant used in these cases (but not some others) by Mathematica as containment sets or cset intervals. The IEEE 1788 [6] standard for interval arithmetic declines to endorse this cset interpretation, but allows that it might be useful.1¡ 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`.

The concept of cset has ∞/∞ represented by the interval $[0, \infty]$ and $0/0 = [-\infty, \infty]$ for example.

Adopting the IEEE interval 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 irresolvable difficulty that might emerge, if, through assignment, some cancelled subexpression might be found later to be infinite. Paradoxical false "proofs" can be constructed with such results. As mentioned earlier, performing even one arithmetic step around infinities may be hazardous.

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 to be used in case `n` is later is given a value.

In the same vein we can also consider `infinity[0,c6]` to be "non-directional complex infinity" [7]. Or just leave that alone unchanged.

Following the implications of this development another consideration turns up: the encoding of a signed zero. Although this is a break with the usual CAS setup, it is included in the IEEE 754 floating point standard, and for symbolic systems 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 possibility if we wish to use a projective model, mathematically speaking. It means, however, there is no separate $-\infty$, probably unacceptable.

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 two reasons: (a) explanation and (b) implementation.

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

[6]`https://standards.ieee.org/standard/1788_1-2017.html`

[7]In Maxima this is `infinity` as opposed to positive real `inf`.

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. The Maxima `limit()` simplifier attempts to cut through some of the complications by complicated heuristics involving floating-point computations.

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 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[8] to handle other special cases over the years. It is also mostly orthogonal to what is proposed in the next two sections.

We have implemented the scheme above, in conjunction with a package for interval arithmetic. There is no need to actually use the interval facilities if the use of indexed infinities is the sole interest.

It is not the only technique we have explored. 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.

6 The Rational Number Alternative

Common Lisp, Maxima, and other CAS as well as some other new programming languages[9]. 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 [10].

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 the simplified form `(rat simp) 1 2)`. In principle an expression could have other "flags" added after the `simp`, but past practice knowing that something is a rational number implied a complete set of its properties[11].

[8]Some would say, encrusted.

[9]e.g. Julia

[10]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!

[11]but see below.

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 than included here. 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. We run out of tricks if we need to represent "unsigned" 0 differently from "positive" 0, so they are the same.

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 overwrites several pieces of the simplifier to incorporate a few very small code segments to accommodate the new rational numbers[12].

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 $1/2 + \infty$ 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[13], 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 software-implemented arbitrary-precision "bigfloats", where there is no limit on the exponent value. Since the `((rat) n d)` structure is widely used, there may be programs that we have not located that implicitly require the denominator to be strictly positive, causing difficulty with the encoding of -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. As a consequence of the new changes at the elemental level, many procedures will produce the newly expected infinite or indefinite results as a consequence of following the usual arithmetic routines.

The interface to floating-point computation is one problem area. Conventionally, most programming languages and CAS follow the convention that combination by addition, multiplication, or other operations of a rational with any floating-point format produces a new float[14]. If that rational is 1/0, we must intercede and either make the result a non-float 1/0, or convert to a float (version of) ∞. Maxima has no bigfloat ∞ to return, so the latter choice is infeasible[15]

Especially in the absence of other mathematical imperatives, Maxima convention looks toward the Common Lisp standard. Unfortunately, given the historical overlap in the standardization efforts for Common

[12]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.

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

[14]In Common Lisp as well as Maxima

[15]For finite floats conversion to a bigfloat is always possible. If the variable `mpdf` is set to the most positive double float, then the computation of `2.0b0*mpdf` produces the bigfloat $3.59... \times 10^{308}$. The computation of `2*mpdf` which requires returning the answer in double-float results in an error: the system signals a floating-point overflow and halts the computation.

Lisp and IEEE 754 floating-point, and the (initially meager) conformance of hardware to the floating-point requirements, only finite floating-point numbers were written into the Common Lisp standard. Full floating-point support in standard-conforming Common Lisp implementations depends on extensions to the language for nuances that can differ between implementations. Maxima runs on numerous different implementations on several popular hardware platforms, requiring either careful compromises, or (rarely) programs that operate differently on different systems. The software bigfloats cannot fix ∞ though.

One possible uniform response would be to refuse to explicitly or implicitly convert to or from any floating-point version of 1/0, or when 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. Thus an explicit recognition and treatment of 1/0 (by an error-handler) could return any required value, including an IEEE infinity.

Compared to the embroidered array of many infinity objects of the earlier section, this approach gives us just one ∞, 1/0 encoded internally as the lisp list `((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 this model there is only one positive infinity, and so 1/0-1/0 must always return the same value (which is 0/0).

Can we fix this? 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. In fact, we view with suspicion any computation that depends on memory locations since it puts us at the mercy of the storage allocation system.

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 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 our current implementation but is feasible.

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 the distributive law fails: $a \times (b+c)$ is not $ab+ac$. As: $\infty \times (3+i)$ would be $3 \times \infty + i \times \infty$ which simplifies to $\infty + i \times \infty$, which is a different direction.

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 of 1/0 to floating-point! While this is a small change, it is one that is repeatedly executed in a main loop of the simplifier[16], this is a loss in speed rather than, as in the gcd situation, a potential speedup since there we 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 may require a similar small patch.

7 The Rational Expression CRE Alternative

The Maxima system has a subsystem for manipulation of canonical rational expressions (CRE), part of Macsyma's original system circa 1967. In today's vernacular this would be call an object-oriented system which defines representations and operations. In essence, each CRE object is stored as a ratio of two relatively compactly-encoded 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.

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

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.[17] 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 ∞ 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 polynomial headers. 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 we 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 effect to the general simplifier.*

Along with this alternative comes some assumption that we have thought through 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 $x = 1/0$.

The answer is yes. In the algebraic structure (officially an algebraic Field), these simplifications are *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 if `x=0` the product is 0. The zero appears "stronger" here. One way of interpreting all 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 these new infinity-related symbols first-class objects *outside* the CRE subsystem. As constants such as `exp(1/3)` are just represented symbolically as a non-rational "kernel" with respect to the CRE form, so is `exp(-1/0)` . At a few junctures it is necessary to look "inside" these kernels, such as a command that will try to "numerically evaluate" an expression. it is plausible that we can construct a route from these new representation to some floats, finite or not. When such conversions are infeasible, we must signal errors. (We must already do so for overflow!) If we can keep the conversion from infinities and NaN limited to the bridge between CRE form and numerical evaluation, we have a potentially simpler implementation route.

7.1 What about just using floats all the time?

If IEEE 754 floating-point arithmetic is available, can we just use floats to encode ∞ and 0/0? We could narrow the focus to one precision, say double-float and see if that can be used for the representation.

[17]The polynomials are stored in a recursive fashion, with the depth keyed to the number of variables. This detail is immaterial to this particular discussion.

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 finite means 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.

Given a supportive floating-point support system this would consolidate the otherwise duplicative notions as floats. Given: two (signed) infinities, two zeros, and a representation (actually many) for indefinite via NaN.

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 a considerable loss of information. After all, most CAS can represent numbers as bigfloats or exact rationals that far far exceed the overflow threshold of the double floats, and thus "floating overflow" could be handled by promotion to "bigfloat" implementations in software, rather than ∞.

This seems to involve too many compromises to the exact and symbolic mathematical model promoted by CAS.

8 Comparison with Other Systems

8.1 Maxima's (c. 1971 - present) system

Since any of the special names used for infinity in the standard system appear as 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`. A patchwork of special cases and checks by particular commands represents the operations supported for infinities, in particular for endpoints of integrals, sums, products, and limits. While this results in certain demonstrable correct results, it is not overall robust, and seeking a more useful outlook motivated this paper.

8.2 Mathematica

Mathematica 13 [18] has `Null, Undefined, Indeterminate, Infinity, ComplexInfinity, DirectedInfinity[]`, and `DirectedInfinity[direction]`.

We have run examples in several versions of this system over the years since 2015, and there have been some changes.

`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." Apparently adding `Indeterminate` to `Undefined`, "a quantity with no defined value" makes a result that is numerical.

The test `Infinity==Infinity` returns `True` and yet

`Infinity-Infinity` is not 0 but `Indeterminate`.

This is also true for other pairs such as `DirectedInfinity[I],I*Infinity`

The test `1/(1/Infinity) ==(1/(1/(-Infinity)))` returns as

`ComplexInfinity==ComplexInfinity`, so it is apparently neither true nor false.

`1/0` returns as `ComplexInfinity`

[18]`https://wolfram.com/`

`-ComplexInfinity` returns as `ComplexInfinity`.
but
`Limit[1/x,x->0]` is `Infinity`
`Limit[1/x,x->0,Direction->-1]` was, in the earlier version 10.10.2.0, `Infinity` but is now `ComplexInfinity`
`Limit[1/x,x->0,Direction->"FromBelow"]`, with a changed syntax, is now `-Infinity`
`Limit[-1/x,x->0,Direction->-1]` was, in the earlier version `-Infinity`, (which was outright wrong),but
`Limit[-1/x,x->0,Direction->"FromBelow"]` is now `Infinity`.
`0*Undefined` is `Undefined`
`DirectedInfinity[]` is simplified to `ComplexInfinity`,
"a quantity with infinite magnitude but undetermined complex phase."
`Arg[ComplexInfinity]` is `Interval[{-Pi,Pi}]`
`Arg[Infinity]` is 0
`Sin[Infinity]` is `Interval[{-1,1}]`.
This is arguably a misuse of interval arithmetic.
`Null-Null` is 0. `0*Null` is 0, `Indeterminate*Null` is `Indeterminate`,
`4*Null` is `4*Null`.
`Infinity+I` is `Infinity`

8.3 Maple

I do not have a current version of Maple [19] but I have looked at the version 18 manual and the online help.

In the manual, the text "undefined" exists in two places. Once describing that, when in the `RealDomain package`, a computed result that would conventionally be a complex value (say square root of a negative number), this would not be real, and hence is undefined. The second location points out that the *bidirectional* limit of $1/x$ as x approaches 0 is returned as undefined. The help system is more forthcoming: $\infty - \infty$ returns undefined, and in a manner somewhat familiar to our proposal above, it is possible to tag a non-finite number: in particular, the conversion of undefined to a float via `Float(n,undefined)` for `n` a non-zero operand, provides distinct undefined values.

It appears that the software `Float` has what we would consider a NaN representation, perhaps encoded analogously to the IEEE 754 floats as some "maximum" exponent. The Maple software floats are base-10.

A different representation is provided via `HFloat()` which is the command to produce a hardware floating-point number, presumably an IEEE float.

There does not seem to be a way to construct an IEEE hardware NaN, analogous to the tagged `Float`.

From comments on MaplePrime [20] I gather that `infinity-infinity` simplifies to 0.

It appears that the tagged infinity proposal above would be relevant to Maple, but we might try the equivalent experiments as shown in Mathematica in Maple 18.

9 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/infinity.lisp` 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.

[19] `https://www.maplesoft.com/documentation_center/maple18/usermanual.pdf`
[20] `https://www.mapleprimes.com/questions/228696-Can-Anyone-Explain-Me-How-infinity`

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, which is why it is in that file.

Neither of these schemes has been entirely integrated into Maxima.

10 Acknowledgments

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

11 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," *J. Symbolic Computation* 39, 5, May, 2005 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," `https://people.eecs.berkeley.edu/~fateman/papers/extrat.pdf`, 2004