

Interval Arithmetic, Extended Numbers and Computer Algebra Systems

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley

October 10, 2009

Abstract

Many ambitious computer algebra systems were initially designed in a flush of enthusiasm, with the goal of automating *any* symbolic mathematical manipulation “correctly.” Historically, this approach resulted in programs that implicitly used certain identities to simplify expressions. These identities, which very likely seemed *universally* true to the programmers in the heat of writing the CAS, (and often were true in well-known abstract algebraic domains) later needed re-examination when such systems were extended for dealing with previously unanticipated kinds of objects by extending “generically” the arithmetic or other operations. For example, approximate floats do not have the mathematical properties of exact integers or rationals. Complex numbers may strain a system designed for real-valued variables. In the situation examined here, we consider two categories of “extended” numbers: ∞ or *undefined*, and real intervals. We comment on issues raised by these two troublesome notions, how their introduction into a computer algebra system may require a (sometimes painful) reconsideration and redesign of parts of the program, and how they are related.

1 Introduction

We discuss methods for dealing with intervals and extended numbers in a computer algebra system (CAS). We think the interval approach presented in section 2 is novel compared to its usual treatment, partly because it is embedded in a CAS. Our approach uses some computations which are relatively easy to program in a CAS, but would require building up much more infrastructure if attempted in the numerical context in which interval arithmetic *usually* appears. Furthermore, the CAS user may be more inclined to allow extra computation time, given that symbolic computation is typically done in a setting where accuracy or correctness is more important than speed. In a CAS it is sometimes possible that results can be computed to *arbitrarily high precision* or when conditions permit, *exactly*. In this interval case, we can benefit from either of these special arithmetics. Once we have adopted such usage, it seems plausible to add somewhat more mechanisms and expense in the algorithms to ameliorate a particular pitfall of interval computation: to keep the width of the interval down. The same kinds of arguments hold for extended “number-like” objects such as ∞ . Whereas the IEEE-754 floating-point standard representations of ∞ and undefined are constrained to be exceptional operands fitting within the floating-point format, a CAS can expand upon the notation—using extra time and space as needed—if the user is seeking more informative results. We consider this subject along with intervals because the results are related: they are both non-traditional arithmetic systems.

To return to the notion of failing “identities” indicated in the abstract, here are two examples that might plausibly be built in to a CAS simplifier: for any expression x , $x - x = 0$ and $x/x = 1$.

When might this not be true? When x is ∞ or an IEEE floating-point NaN or an interval, say $[-1,1]$, other rules apply. Yet if a CAS generally failed to apply such common simplifications, even a trivial sequence of symbolic manipulations could result in an expression that is unnecessarily complex and unwieldy. A path must be found that allows only legal operations on extended expressions, or expressions which contain variables which can assume values in this extended domain.

Let us use the term *non-nullable* for some x which is a value or a *pseudo-value* from our extended domain for which $0 \times x$ cannot be replaced by 0, $x - x$ cannot be replaced by 0, and x/x cannot be replaced by 1. A pseudo-value might be $\pm\infty$, (unsigned) ∞ , one of the IEEE floating-point operands `nanSingle`, `nanDouble`, `nan`, or extra items like *indeterminate*, *undefined*, and in many contexts, *real intervals*. How these are notated in a system can vary: it might be convenient to use the notation ∞ or $1/0$ for the same concept, and NaN or *undefined* might be \perp . There might also be such concepts as bounded but not definite, or “no number at all” as would be the case for the empty intersection of two disjoint real intervals.

In some CAS designs, in particular Maple and Mathematica, the designers have yielded to the temptation to use intervals as sets. For example, they compute $\lim_{x \rightarrow \infty} \sin(x) = [-1, 1]$. This is a distinct interpretation of an interval. Ordinarily such an interval is a representation of a particular point along a real-line segment. We can't tell which point. The interpretation of $[-1, 1]$ as this limit is different, being *not* any such point, but something like “there is no particular limit but the function is bounded strictly between -1 and 1”. The usual definition of a limit requires that the distance between a point $-1 < r < 1$ and an interval $I = [-1, 1]$ should be 0, when in fact interval arithmetic tells us the distance is not zero, but $[r - 1, r + 1]$. Since the same limit program (in Mathematica 6) returns $[-2, 2]$ for the limit of $2 \cos x \sin x$, apparently the boundaries need not be tight. In fact, $2 \cos x \sin x = \sin(2x)$ and so tighter boundaries would be $[-1, 1]$. Once one admits non-tight boundaries, it seems it would not be “incorrect” to return $[-\infty, \infty]$ for *any computation of a real limit*. This is unlikely to receive full credit on a calculus exam.

There are further ramifications: computing $\sin^2 \infty$ as $[0, 1]$ in Mathematica suggests that in some obscured form we might start with something equivalent to $\sin^2 x + \cos^2 x$ but a failure to sufficiently simplify followed by an evaluation would ultimately reduce this to $[0, 2]$. This may be acceptable in some circumstances, (like if you find it acceptable for $\lim_{x \rightarrow \infty} 1 = [0, 2]$.) but it points to a need for the user or the program controlling the computational flow to check for intervals or unconventional operands pervasively, *and not only whenever limit calculations are used*. This may slow down ordinary computations on ordinary objects, for which such shortcuts as “0 times anything is 0.” almost always prevails. Indeed, using such rules is hard to avoid unless one can carry along a computation with an object like “0 unless x is a \perp ”.

A more pervasive issue is the alteration of the limit program from an equivalence transformation (ET) to an approximation program (AP). By an ET we mean a program that changes one form¹ to another without changing its value (*perhaps*) with some singular exceptions. Ordinarily the idea behind an ET is to provide an equivalent but, by some measure, simplified expression. Thus an ET simplification program might reduce $\frac{2}{\sqrt{2}}$ to $\sqrt{2}$ or x/x^2 to x . In the latter case we have found a form which is the same². We expect all routine simplification transformations to be ETs, although we may be occasionally disappointed when they are not. On the other hand, an AP would be a program that given an expression, may produce (a) an approximate numeric answer, where the request to obtain such a numeric result is signalled implicitly by inserting a floating-point number as in $\sin(3.0)$ which may result in the approximation 0.14112000805987 or (b) an explicit request for a numerical valuation, or (c) a symbolic approximation such as a request for a Taylor series approximation:

$$\sqrt{x+1} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \dots$$

While no one legislates such matters, we expect that ETs follow certain rules, like ET(a)-ET(b) is equivalent mathematically to some ET(a-b), assuming appropriate domains.

Ordinarily a limit calculation would be an ET. This assumption is violated if it returns \perp or an interval. Arguably, then, if we want `limit()` to be an ET, it should return “unevaluated” when it would otherwise be \perp or an interval. This would allow certain calculations to proceed by (for example) allowing limit expressions to contain their original information in such a way that they might be combined with (or cancelled by) other related limits which might otherwise also have infinite values.

As an aside, it is possible to group other commands in a CAS into two more classes:

- Valuation transformations (VT) in which an expression is evaluated by substitution of values for indeterminates, e.g. $x = 1$. Numerical approximation could be viewed as a VT where the values are approximate numbers, but some valuations are exact or even symbolic as $x = \cos(\theta)$.

¹A more careful formulation would worry about domains. Here we are, with some trepidation, asserting the domain is “whatever the CAS knows about”.

²except when $x = 0$ (arguably).

- Derived transformations (DT) in which some new expression (non-equivalent) is derived, perhaps by differentiation, integration, or simply by applying arithmetic operations and other combinations to expressions.

(There are also input and output commands for reading, writing, plotting, etc.)

2 A closer look at intervals

Let us revisit that assertion that $[-1, 1] - [-1, 1]$ might be zero, or not. There is a dependency issue, well known in interval arithmetic (or “reliable computing”) circles, in which the inability to determine the “source” of an interval, i.e. whether two intervals are correlated, leads to wider (and therefore less useful) results.

In our example, if the expression means the set $\{x - x \mid x \in [-1, 1]\}$ the answer is 0. If we mean $\{x - y \mid x \in [-1, 1] \text{ and } y \in [-1, 1]\}$, then the answer is the interval $[-2, 2]$.

Is there a neat way of distinguishing the two cases? Here’s a possibility (using the CAS Macsyma), where these might be notated

```
block([x:interval(-1,1)], x-x)
\medskip
```

and

```
block([x:interval(-1,1),
      y:interval(-1,1)], x-y)
```

This locution is an attempt to force a kind of interchange of simplification and evaluation, where in the first case the expression $x-x$ would be simplified to zero and in the second case the expression $x-y$ would not be simplified, but just evaluated using interval arithmetic rules³.

Another example which illustrates the possibility of getting tighter bounds with additional work, is the expression $\sin(x) \cos(x)$ which, on the face of it, for all real x , seems to be in the interval $[-1, 1]$. In fact, the expression is in $[-1/2, 1/2]$, a result which becomes clear by recalling that $\sin(x) \cos(x) = \sin(2x)/2$.

A non-constructive definition of interval arithmetic can be phrased in terms of constraints on a function. Elucidating the situation of dependency requires more than one variable, so let us start with two. Then given a real-valued function of two real-valued arguments to evaluate: $f(x, y)$ for x, y interval-valued between $[x_0, x_1]$ and $[y_0, y_1]$ respectively, we return an interval $[z_0, z_1]$, where the real-valued z_0 is the minimum of $f(x, y)$ with the constraints $x_0 \leq x \leq x_1$ and $y_0 \leq y \leq y_1$, and the corresponding maximum is z_1 . While the interval $[z_0, z_1]$ then is the “best” answer, it might not be possible to find it conveniently either because we have lost information during the calculation, or the minimization computation is computationally too costly. We tend to accept the computation of some other, looser, version: $[w_0, w_1] \subseteq [z_0, z_1]$. That is $w_0 \leq z_0$ and $w_1 \geq z_1$. We may also need to extend the interval notation to allow for situations such as division by zero.

Much effort can sometimes be expended in improving “tightness” of the result, and this may be useful in the long run, since any slack in the result can be magnified by subsequent operations.

In our assumptions about a CAS, we mean by “interval()” a program which produces a new object, not shared.⁴

³Without changes in Macsyma, which currently does not have rules for arithmetic about intervals, it doesn’t work. They both return 0.

⁴Maple has a feature making all compound objects which simplify to the same structure isomorphic. Implementationally they are pointers to one copy in memory. This is helpful in some processing, but not this one. There is sometimes a difference between holding two chocolate-chip cookies, one in each hand, and holding just one cookie but with both hands.

3 An approach

In this section we discuss a collection of approaches for computing with *real intervals*. We claim no novelty in these matters⁵. Our perspective here is to briefly introduce ideas to a reader who may not have seen these before, or has not seen the alternative possibilities to ‘naive’ intervals. Our objective is to provide enough background so the reader can seek out further information in the references, but also so that we can reasonably, in the following section, point out the relationship with pseudo-values that occur in CAS.

3.1 Intervals are Three-tuples

We create a numeric interval of the real line by calling a function, `Ninterval`. For example, for the interval between -1 and 2, we evaluate a form like this `v=Ninterval(-1,2)`. What this function constructs, rather than our usual representation of an interval as a pair, e.g. `[-1,2]`, is an internal data structure with *three* slots. The first two are the visible upper and lower limits, often denoted \underline{x} and \bar{x} , where in our example, $\underline{x} = -1$ and $\bar{x} = 2$. These can be exact integer or rational numbers, machine floating-point numbers, software “bigfloats”, or (though we do not recommend this) literal parameters or expressions (say a or $\exp(b-1)$). Before constructing the third slot, we must first manufacture a new symbolic variable name⁶. Here we denote that variable by `#n` for some integer `n`. The third slot is then an algebraic expression involving that variable, initially just `#n`. The exact representation used is specified in the implementation, and might be a list of polynomial coefficients, a “tree” or some other form that might be convenient in a CAS.

Our example v might be described this way: `[-1,2, #5]`. However, since we (humans) ordinarily do not need to see the extra slot, programs ordinarily will not display it in the computer output. For purposes of exposition in this section of this paper, we will show them.

If we compute $u = v \times v$, we create the slots in u as `[0,4,#52]`. Note that if we had two independent intervals that happened to have the same bounds, and multiplied them, we would compute `[1,2,#5] * [-1,2,#6]` as `[-2,4,#7]`. This is the interval `[-2,4]`, which is wider than `[0,4]`, and where we have generated a new variable, `#7` which means we have lost some information about the origin and possible values of this interval, if, say, we were to further combine it with `#5` or `#6`.

Let us continue our example. If we compute $u \times v$, we can notice a coincidence in indexes in `[-1,2,#5] * [0,4,#52]`. By realizing that we are computing `#53`, the resulting interval is `[-1,8]`, instead of the result of straight multiplication of intervals, which would give `[-4,8]`.

Under certain conditions we can maintain this stream of computation, and more generally recompute the expression in the third slot. To do this we must not only represent the polynomial, but remember the original value. In this example we do this by keeping the original `#5` around as long as there is a live “accessible” value that uses it. There is a convenient mechanism that is built-in to Lisp for this purpose, a “weak hash table”.

Another optimization we can make is for quadratic polynomials. We can rewrite $p = 3x^2 + 2x + 1$ as $q = 3(x + 1/3)^2 + 2/9$. This is a “single use expression” referring just once to x , and provides tighter bounds as $p([-1.1, -1.0])$ is `[1.8, 2.63]` but $q([-1.1, 1.0])$ returns the tighter (in fact optimal!) `[2.0, 2.43]`. The order of computation here is based on “completing the square” but unfortunately does not generalize to higher order.

3.2 Variations

What should we allow as acceptable expressions for that third slot? We initially considered just expressions of the form rx for a scalar number r or rx^n for some integer n but then realized that it would be simple to use existing subroutines for symbolic mathematics to carry around a more elaborate form. The next elaboration was a polynomial in (any) *one variable*, in some encoded canonical form. Given such a form, we can feed it into a program for further analysis: for evaluating a polynomial P in one variable x at an interval point, the tightest bounds can be computed by considerably more elaborate computation than just going through the adds and multiplies: locating suitable intervals around the zeros of $P'(x)$, then evaluating P at those places

⁵Indeed, any time we thought we had a new idea in interval arithmetic, we found we were simply rediscovering something that had been previously identified as interesting (and often analyzed in great detail).

⁶a counter is incremented after each newly constructed name.

to find relative extrema. Combined with P evaluated at the endpoints (absolute min/max), we can find sharp bounds subject only to the representation of the endpoints. Such a computation *could* be worthwhile if the advantage of finding tighter bounds is more important than the considerably increased computational expense.

Any expression that turns out to be more complicated than we wish to deal with can simply be given a fresh name, and from that point forward it is assumed to be independent of all previous variables. Thus our current program would never notice the correlation possibility that is inherent in the identity $\sin(x) \times \cos(x) = \sin(2x)/2$ since each of the expressions $\sin()$ and $\cos()$ would be given different (presumed to be independent) names. This would lead to more conservative bounds, but not incorrect ones.

Where do we draw the line, though: what is too complicated? Can we in fact use something more elaborate than polynomials? Indeed, there are a host of variations on interval arithmetic: our invention of a “polynomial” slot is subsumed under any of a number of previously documented innovations, many of which are reviewed by Arnold Neumaier, see <http://www.mat.univie.ac.at/~neum/papers.html#taylor>. This paper reviews so-called Taylor forms—an extension which revolves around carrying a truncated Taylor series with remainder, evaluated at the midpoint of the interval. In certain calculations this additional information can be effective in reducing the width of the intervals being carried. To produce this model, we could use the extra slot effectly as an expression of a Taylor series, with remainder, for the number being represented. We could start our trigonometric calculation by considering that we are evaluating $f(x) = \sin(x)$, where $x = [a, b]$. We then expand \sin in a Taylor series around the midpoint, $x = x_0 = (a + b)/2$, and use an expression for the remainder (error). Most CAS provide a suite of operations on Taylor series, and so can implement this “off the shelf”. The operations are similar to those on polynomials with the major difference that the Taylor method sets a global order, (maximum degree) and high-order terms are truncated. The error term is calculated separately as another interval calculation.

It is important to observe that the evaluation of the remainder term, which can itself be elaborate, may have to be done carefully too, or some blow-up is re-inserted. Programs for evaluating derivatives can be generated by “Automatic Differentiation” programs, but these must also be evaluated in an interval fashion. When the error term looks sufficiently well-behaved, “naive” interval arithmetic may do. Illustrations of the payoff from maintaining tighter bounds seem tied to a procedure which can terminate sooner because of the bounds. A typical case may be a Newton iteration combined with interval bisection which may converge with *far fewer iterations* and thereby pay for for the more expensive operations.

Other possibilities include allowing the extra slot to contain a linear function of any number of interval values (more about this later). In this case a linear-programming method could be used to find min and max. If the extra slot were allowed to be a multivariate polynomial, a method based on cylindrical algebraic decomposition might be suitable.

An idea called “Affine Arithmetic” for intervals has also been pursued in the recent literature on reliable computation. In this model, a quantity x is represented as a first-degree (“affine”) polynomial in the values it depends upon. In particular,

$$x = x_0 + x_1e_1 + x_2e_2 + \cdots + x_ke_k$$

where the $\{x_i\}$ are given “real numbers” and the $\{e_i\}$ are dummy variables, assumed uncorrelated for calculation purposes, but each of whose values is known to be in $[-1, 1]$. When possible the representation is maintained through ordinary arithmetic, and when it is not, the operational routine must find an alternative acceptable approximation by introducing another variable e_n and positing it as uncorrelated. Affine arithmetic (AA) seems especially appealing when geometric objects are defined in which the coordinates share a set of dummy variables. This situation produces centrally-symmetric convex polygons or “zonotopes”. In two dimensions these can be visualized as “lozenge” shapes that are smaller than the enclosing rectangular intervals that would appear from naive intervals. There are benchmarks illustrating particularly advantageous results in approximate computations for graphics: they can provide closer adherence to lines or planes that are being approximated. Not all computations ultimately benefit from AA: it may be beneficial to switch between methods to increase overall efficiency: AA may be better for small intervals. An extensive comparison of interval variations is given in the paper by Martin *et al.*[2].

See for example,

<http://www.icc.unicamp.br/~stolfi/EXPORT/projects/affine-arith/Welcome.html> or
<http://www.tecgraf.puc-rio.br/~lhf/ftp/doc/oral/aa.pdf>

Implementing AA on top of our structure is straightforward, and we have done so up to addition and (non-trivial) multiplication, for arithmetic combining scalars and intervals.

Yet another possibility is to try to compute an algebraic “single use expression” (SUE) from some set of variables. (see whitepaper on Sun Forte Fortran by G.W. Walster <http://developers.sun.com/prodtech/cc/products/archive/whitepapers/tech-interval-final.pdf>). This would transform an expression like $ax+bx$ where a and b are scalars, but x is an interval, into $(a+b)x$, which may be a tighter expression.

Similarly, for intervals x and y , $A = x/(x + y)$ can be rewritten as the SUE $B = 1/(1 + y/x)$, although the acceptable domains for x and y have now been changed. (Look for division by interval expressions that might include zero!) Walster suggests obtaining the answer by computing $C = 1 - (1/(1 + x/y))$ and then returning their intersection, $B \cap C$.

One approach for SUE is for us to establish a set of heuristics for driving an algebraic expression into a more-nearly SUE form. A good algorithm to find an optimal “sharp” solution (that is, more efficient than exhaustive enumeration and evaluation) would be welcome, but tradeoffs such as whether a SUE version with respect to x is better than a SUE with respect to y , or if some other combination is yet better, cannot be computed at compile-time, since in general the optimum depends not only on the expression (which is presumably available at compile-time), but on the particular *run-time* interval values associated with x and y .

A possible, but somewhat unappealing, approach can be constructed around multiple arrangements of the same expression. Given any collection of “mathematically equivalent” expressions transformed at compile-time into various interval expressions, each version could be numerically evaluated, and the tightest solution emerges by computing their intersection. Such evaluations must be done while aware of the possibility of the introduction of extraneous singularities; if any of the expressions is non-singular at a point in the interval, then it presumably reflects a valid mathematical result. A good expression might be a Taylor series expansion at the midpoint of the interval. This cannot be computed without knowing the interval endpoints.

We could attempt to use some existing (essentially black-box and possibly heuristic) programs like Mathematica’s `Maximize` and `Minimize`. Yet, even when they nominally work according to their own specifications, the answers may not be satisfactory for interval computations. Reliable computing requires strict inclusion: these imply stronger requirements for directed rounding of endpoints than typical numerical min/max programs.

In Nedialko et al [7], various methods are considered. A theoretical comparison can be summarized by an optimality result: representation by polynomials is, in their presented context, optimal.

3.3 Other approaches to improved interval computation

Other techniques, such as (repeated) bisection of intervals can be used to attempt to narrow the result width. Our suite of programs includes this conceptually simple technique: the result for $P(x)$ would be the smallest interval containing all the ranges of $P(x_1), \dots, P(x_n)$ where x has been subdivided. That is, x is the union of the x_i ; some criteria determine which of the subdivisions to divide again in an attempt to narrow the interval. (In particular the subdivisions containing the current maximum and minimum should be re-divided.)

This kind of interval arithmetic is more encapsulated and therefore “hands-off” compared to the usual numeric version. It takes the form of a functional programming model: new values may be created out of components; but we do not allow user-level operations like “change the lower limit of variable $x = [\underline{x}, \bar{x}]$ to be the number 43.” Instead, we allow the creation of a new object which can be re-assigned to the same name: $x = [43, \bar{x}]$. Old values, if they are no longer accessible, will be collected, and their memory locations re-used.

4 Pseudo-values: Infinities and intervals

At this point it may not seem that there is any direct connection between intervals and extended numbers such as infinity. In fact, the resemblance has more to do with the techniques to resolve difficulties than the mathematical concepts.

Note that without intervals, the production and therefore occurrence of infinities is limited to singularities (of which division by zero is the most obvious, but not the sole example). With intervals, the exceptional treatment of zero must be extended in some way to *intervals that contain zero*. Once having produced such an object, can one make it go away? For example, can we be assured that $1/(1/x) = x$ true, even if x is an interval containing zero?

4.1 Infinity as a symbol

One of the problems of a CAS design is that there is a tradeoff between convenience for human users and for computation. As an example, computationally, we might construct a system in which an infinite value appears only as a result of certain operations, and in each case we can find some more informative alternative. Unfortunately, if its origin is simply “a human typed it into the system” we do not have an alternative to “just compute with this.”

What can we propose? We can keep $\infty - \infty$ from becoming zero if, each time we generate (a new) ∞ , we give it a unique subscript or index, in a manner similar to the indexed variables or third slots used with the intervals. That is, $\infty_1, \infty_2, \dots, \infty_n$. In Macsyma/Maxima this is rather simple to start: `(inf_counter:0, infinity):=(?incf(inf_counter),Infty[inf_counter]), nofix(infinity)`

After this, any mention of `infinity` as in `infinity+3` will generate an expression like `Infty[4]+3`. In \TeX this could be displayed as $\infty_4 + 3$.

This is not enough: A search through the Maxima code for all producers and consumers of infinities may reveal locations that must be changed to conform to this model. These occur principally in the `limit`, `sum`, and definite integration facilities. Also saving such values off-line and reading them back must re-create the proper values, shared when appropriate, and distinguished from values created from other processes.

We need a new simplifier that is able to compute a kind of pessimistic simplified form. For example, given the expression $\infty_2 - \infty_3$, *assuming that there are no occurrences of ∞_2 or ∞_3 anywhere accessible in the CAS*, can be simplified by generating an undefined: `und`. Such symbols are also indexed. If there are other occurrences of those ∞_k symbols elsewhere, it is prudent to keep the expression uncombined in the hope that at some later point one of those symbols will interact and cancel. We typeset the undefined quantity with the “bottom” symbol, for example, \perp_4 . This gives us an opportunity to consider a case where we assign `x := limit(1/t^2,t->0,plus)` and thus `x= ∞_5` , a specific value. Then `x-x` is $\infty_5 - \infty_5$, and indeed this can be simplified to 0 because this is the difference of two of “the same ∞ ”.

Mathematica uses a notation for infinity which provides a direction in the complex plane; $(1+i)\infty$ is effectively `DirectedInfinity((1+i)/sqrt(2))` after normalization. There is presumably only one infinity in each direction, and so rules combining them must either be very conservative or sometimes wrong. This design suggests that for each directed infinity generated, there should be a subscript as well.

This simplifier must deal with any expression with one or more infinite or undefined objects, possibly other forms such as “indefinite” (but bounded) and generally should come up with the most informative and compact result possible. For example, *in the absence of any other occurrences of the same-indexed infinities*, $\infty_2 + \infty_3$ could be replaced with ∞_4 as a simpler result⁷. In similar circumstances (no dependencies) there is no point in retaining the details of $(\perp_4 + 1)/\perp_7$. This should be replaced by (say) \perp_8 . An expression involving such objects should be simplified as much as possible, using standard rules for combination. This is not necessarily trivial, and for rational functions may be solved using a resultant technique devised by Neil Soiffer in his MS project at Berkeley (1980). That work calculated the minimal number of “arbitrary constants” needed to provide the same degrees of freedom as a given expression with perhaps many such constants.

5 Algebra

This section raises questions, but does not answer all of them. The approach taken by a particular CAS may be delimited by the activities it is supposed to support. Thus a base for a theorem-proving system might

⁷To reiterate: it would be a mistake to make this replacement if there were another expression algebraically dependent on either of the two component ∞ objects, since the replace would lose dependency information.

be more “pedantic” than one intended for (say) checking hand computations where it is known beforehand that extraneous solutions will occur and can be discarded.

We start by observing the properties that are consistent and those that fail when we have \perp and ∞ without subscripts. (or in some cases, with subscripts).

Consider that at the base of the ordinary simplifier we assume that the otherwise uninterpreted symbols x, y, z etc., come from the properties of integral domains, which include the postulates for commutative rings.

Commutative Rings:

1. Closure: a, b in R imply $a + b$ and $a \times b$ are in R .
2. Uniqueness: $a = a'$ and $b = b'$ imply $a + b = a' + b'$ and $a \times b = a' \times b'$
3. commutation: $a + b = b + a$; $a \times b = b \times a$.
4. Association: $a + (b + c) = (a + b) + c$; $a \times (b \times c) = (a \times b) \times c$.
5. Distribution: $a \times (b + c) = a \times b + a \times c$.
6. Additive Identity (Zero). there is an element, 0, such that $a + 0 = a$ for all a in R .
7. Multiplicative Identity (One). there is an element, 1, such that $a \times 1 = a$ for all a in R .
8. Additive Inverse: $a + x = 0$ has a solution x in R for all a in R .

integral domain adds

9. Cancellation: if c is not zero and $c \times a = c \times b$, then $a = b$. (*fails for \perp and ∞*)

Other property usually assumed for the symbols include *ordering*.

10. Trichotomy: $a > 0$, $a < 0$ or $a = 0$. (*fails for \perp*)
11. Transitivity: $a < b$ and $b < c$ implies $a < c$. (*fails for \perp*)

Here are some consequences:

12. $a < b$ implies $a + c < b + c$. (*fails for \perp and ∞*)
13. $a < b$ and $0 < c$ implies $ac < bc$. (*fails for \perp and ∞*)
14. $a < b$, $a = b$, or $a > b$. (*fails for \perp and ∞*)

What about fields, e.g. reals?

15. A field is an integral domain in which each element e except 0 has an inverse, e^{-1} such that $e \times e^{-1} = 1$. (This seems simple enough, but has lots of further consequences.) (*fails for \perp and ∞*)

The failure of elements ∞ and \perp to satisfy some of these postulates either invalidates certain standard rules of operation in a CAS or on the other hand limits the scope of a CAS. Since the ambition of at least some CAS is to encompass all of mathematics, such systems must either entirely march \perp out of mathematics, or “do something”.

We would like to know that, if we use rules such as $0 \times x = 0$, $x/x = 1$ and $x - x = 0$, that these statements are true for any possible legal assignment of values for x .

Since these are not true for certain values of x , what should we do?

Consider clarifying =.

The usual way of providing inference rules for equational logic, includes the following:

Substitution: If P is a theorem then so is $P[x \rightarrow E]$ where $P[x \rightarrow E]$ means textual substitution of expression E for variable x in P .

If $P = Q$ that is, P and Q are of the same type and are equal, then $E[x \rightarrow P] = E[x \rightarrow Q]$

If $P = Q$ and $Q = R$ then $P = R$.

Some CAS essentially say that two expressions are “=” under much looser conditions. For example

- a. They are equal if, for *nearly every* substitution of values into the variables, the expressions evaluate to equal expressions. Yes, this “nearly every” sounds just wrong, but the argument is made by the authors of Mathematica that a “generic solution” is acceptable if all exceptions to that solution are in a space of lower dimensionality. Example: $x/x = 1$ even if it is perhaps false if $x = 0$ (That’s only one place) or if x is an interval containing zero (That includes many intervals). Another example: For all pairs of real numbers $\langle x, y \rangle$ in R^2 , $(x^2 - y^2)/(x - y) = x + y$ except along a particular line, namely $x - y = 0$. Such an argument defeats the premise of high confidence in the capability of a CAS program to produce “guaranteed” answers. In particular, it is a hazard if one proposes to use a CAS prove other programs correct⁸.
- b. Two floating-point single- or double-precision results are equal if they are “close enough” (in some relative or absolute error sense).
- c. Two approximate or imprecise numbers are “close enough”: In Mathematica we can construct a very imprecise number n , `n=N[1,1]`, a number 1 with one digit of accuracy. Then this equation is “true”: $3 + n = 4 + n$.
- d. Mathematica claims that two values each `Infinity` are equal, yet the difference of them is `Indeterminate`. This can’t be right.

How can we reconcile the different behaviors of a CAS resulting from such designs with the intention that a CAS can “prove” mathematical theorems? If a proof that $f(x) - g(x)$ is zero (or is non-zero) “for all x ” (where “prove” is implemented as simplification), is contradicted for some $f(c) - g(c)$ at a specific value c ? Certainly if c is not in the domain of x we can make some case that this does not matter. What, though, if it is in the domain?

5.0.1 What is equality?

If we approach this question operationally from the perspective of a CAS design, the most plausible definition is to say that two objects are equal if a simplifier program provides the same canonical result, that is, lexicographically identical, and programmatically indistinguishable⁹, for each side of the relation.

In Lisp this is `equal`. Lisp has `eq` to represent equality as “same storage location”, as well as several specialized equalities for numbers (`=`), strings (`string=`), and combinations (`eq1`). It is possible to have a canonical expression which is, for example, `3*inf[3]+14*inf[7]`. which would be different from `14*inf[7]+3*inf[3]`; a canonical simplifier would rearrange one or the other so they would look the same. Mathematics texts are generally insufficiently pedantic to make the subtle distinctions necessary for programmers.

We can require that testing for numerical equality, as between floating-point numbers, use a different predicate. (Advice given to novice programmers dealing with floating-point computation is to never test for equality; this advice is overkill: there are occasions when testing equality does make sense.)

Equality for numbers expressed as intervals has several different variations—possibly, certainly, impossible.

5.0.2 An alternative approach

Assume that manipulation in the general simplifier is bound by rules of computation in a commutative field with identities 0,1 with algebraic extensions for each independent indeterminate with domain Real or Complex numbers, and that in this field, $x/x = 1$

Any computation that we do must first make sense in this domain. After the computation is complete, a valuation can be done in some other domain in which an algebraic expression is associated with an ordered list of arithmetic commands that can be performed on elements in any domain, including elements that are infinite, undefined, intervals, or other objects.

⁸We understand why some people advise against learning mathematics from a physicist.

⁹We may (and will) forbid examining the representation at some abstraction level, otherwise we would be faced with conundrums like declaring as unequal two integers because they are stored in different memory locations, even if they had the same value.

Except for the discomfort caused by $x^0 = 1$ and $0^x = 0$, this might work.

consider 8, inverse. $\infty + x = 0$ has no solution. However, if we have a indexed set of infinities, $\infty_1, \infty_2,$ etc. then each ∞_k has an inverse. Real intervals spanning zero arguably do not have inverses unless the language of intervals is expanded to exterior intervals.

consider 9. If $c=\text{inf}$, then $ca=cb$ becomes $\text{inf}=\text{inf}$, or $\text{und}=\text{und}$, which does not imply $a=b$. If $c=\text{und}$, then $ca=cb$ becomes $\text{und}=\text{und}$.

consider 10: und has an unknown sign.

consider 12, 13. the case $c=\text{inf}$ or und does not work.

consider 16. und does not have an inverse. inf does not, since $\text{inf} * 0 = \text{und}$, not 1. Arguably, indexed inf , maybe.

How do intervals fail to satisfy these postulates? First, what intervals are allowed? $[\text{real}, \text{real}]$, $[\text{real}, \text{inf}]$, $[\text{und}]$, $[-\text{inf}, \text{real}]$, $[-\text{inf}, \text{inf}]$

Consider clarifying =

Two intervals are equal if a simplifier program provides the same canonical result for each side of the relation. This can happen if two intervals are each degenerate (one point), and that point is the same number, or if the two intervals match in upper/lower bounds and index expression. Not that in particular it is insufficient to have equal endpoints alone.

consider 8, inverse. $[-1, 1] + x = 0$ has no solution. However, if we have a indexed set of intervals, $[-1, 1, p]$, then $[1, -1, -p]$ may work as an inverse.

consider 9. If $c=\text{inf}$, then $ca=cb$ becomes $\text{inf}=\text{inf}$, or $\text{und}=\text{und}$, which does not imply $a=b$. If $c=\text{und}$, then $ca=cb$ becomes $\text{und}=\text{und}$.

A list of a few plausible rules shows that intervals and the pseudo-values of ∞ and \perp relate to each other. We are here concerned only with *real* values and pseudo-values, and we may need to have separate rules for *complex* \perp . This might occur if one computes $\sqrt{[-4, -1]}$ which has no real value, and could arguably be \perp but perhaps should be $[1, 2]i$ in the context of a CAS which knows about complex numbers.

What is $\sin(\perp)$? $[-1, 1]$ if \perp is *real*, otherwise \perp .

What is $\exp(\perp)$? Perhaps $[0, \infty]$ if \perp is *real*?

Is $[-\infty, \infty]$ the same as *real* \perp ?

Is $[\perp, \perp]$ meaningful? Is it \perp ?

Et cetera.

A CAS designer has a choice to make: are these activities to be confined to the numeric type hierarchy (an uncomfortable choice, I think, in which case there may be both single- and double-float infinities, and the choice of projective or affine models of the real line are relegated to the numeric system), or are such pseudo-values used as an overlay of a separate algebraic system that coexists more with the language of variables and constants (like π and e). The latter seems more plausible in context of a CAS.

An extended discussion of the consequences of choices to make seems to require some subjectivity and consideration of applications, and is beyond the scope of the present paper; a doctrinaire approach is likely to annoy at least some CAS users. We defer further discussion to a future paper and implementation.

5.1 Big-Oh notation

We hesitate to step into this muck, but one might be tempted to relate this material to Big-Oh notation in asymptotic analysis. In particular $O(x) - O(x)$ is not zero, but $O(x) \times O(x)$ is changed to $O(x^2)$. The common notation is further hindered by the misuse of the symbol “=” which is usually considered an equivalence relation. Yet $T(n) = O(n^2)$ does not mean $O(n^2) = T(n)$; an expression which would be rejected as illegitimate. Proposals to remedy this notation by using curly inequalities or set notation have not driven out the traditional notation. It would be curious if implementation in a CAS can provide sufficient motivation to generate an adherence to notational rigor on mathematicians.

6 Implementation notes

For an arithmetic package in which there are typed objects that look explicitly like tagged numeric intervals, it is not difficult in principle to attach methods for all the component-pair combinations that may be of interest. For example, a two-argument “+” in which each of the arguments is a scalar machine double-float can be translated fairly easily into an operation. Running through the plethora of additional operations, e.g. “+” of an interval with an exact rational number, or a symbolic expression like “f(x+1)” can be tiresome, and inheritance hierarchies cannot make all the right decision automatically. Indeed, humans can easily argue as to what is the “correct” coercion. Additionally, some operations may not make sense outside the interval context. For example, there are many more ways of “comparing” intervals: which may involve inclusion, overlap, disjointness. These can be phrased as “certainly” or “possibly” as in “certainly greater than”. etc.

Using the Common Lisp Object System (CLOS), we define methods for

```
(defmethod two-arg-+ ((u Ninterval)(v Ninterval)) ...)
```

etc. for all argument type pairs.

This provides a default for each of the possible comparisons used to evaluate $x + y$. Additionally we can define how to combine an interval with a complex number, a matrix, or a symbolic variable (or perhaps more likely, refuse to do so). In an object-oriented hierarchy, it is possible to place intervals in a lattice with respect to some operations and use inheritance for method resolution, but in some sense a response like “no method available for two-arg-+ with types Interval and Matrix” will wait in the background for unimplemented (or impossible) combinations. The structure for extending the ordinary Lisp arithmetic to intervals (namely by overloading existing n-ary operations like “+” and “*” by using two-argument versions of these operations) is straightforward and can be done partly at compile-time through macro-expansion¹⁰.

As indicated earlier, using CLOS for keeping track of types does not resolve how to integrate interval “stuff” into a CAS, exactly. This is because we must patch an existing system, say Maxima, so that it will not apply identities prematurely that are incorrect for intervals. To illustrate how pervasive such errors can be, note that Intervals were introduced in Mathematica in version 3, and even after many years, in version 5.1, if you type `Interval[a, b] - Interval[a, b]` you get 0. which is presumably false. It has been fixed in version 6.0

Ideally we have localized a simplification that says $x + (-1) \times x = 0 \times x = 0$.

This would operate only in the case that x is nullable. In particular x being some version of infinity, or some as-yet-unspecified interval, makes this simplification false. We must assert some criterion for x before establishing this exception to the general simplification. We must also process x to some extent to see if this criterion holds. It is rather commonplace to try to “short circuit” processing by asserting that if any factor in a product simplifies to zero, the rest of the product need not be evaluated; this leads to inconsistencies.

A choice remains as to whether we should alter the library of programs that constitute the “rational function” library or not. We could alternatively argue that these programs are doing arithmetic in a well-understood field, and we should not use that library if we expect to be violating the axioms for the ground field of the library, say by evaluating at an illegitimate point. We could try intercepting any explicit setting of a rational-variable value to a pseudo-value, but that would not prevent manipulation of arbitrary symbols, where only later the value is proffered. Carrying along the assumptions with the expression (e.g. the answer is “3” but only if x is finite), is a possibility which may be appealing in a theorem proving context, but contending with the exponential growth in such added conditions on expressions is daunting. While there does not seem to be a simple solution, at least recognizing the hazards may make the design of CAS more resilient to the introduction of such extra values.

7 Acknowledgments

A version of this paper was linked to a message on the newsgroup sci.math.symbolic in June, 2006, which provoked a number of useful comments. Thanks especially to Christopher Creutzig. It has been revised as recently as October, 2009.

¹⁰see generic arithmetic (in lisp/generic directory).

References

- [1] Berz-Taylor model of intervals *Reliable Computing* 4: 83-97.
- [2] Ralph Martin, Huahao Shou, Irina Voiculescu, Adrian Bowyer and Guojin Wang, “Comparison of interval methods for plotting algebraic curves,” *Computer Aided Geometric Design*, Volume 19, Issue 7, July 2002, Pages 553-587.
- [3] W. Kahan, “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?” <http://http.cs.berkeley.edu/~wkahan/Mindless.pdf>
- [4] Hongkun Liang and Mark A. Stadtherr, “Computation and Application of Taylor Polynomials with Interval Remainder Bounds,” <http://citeseer.ist.psu.edu/cache/papers/cs/8022/>
- [5] <http://www.nd.edu/~markst/la2000/slides272f.pdf>
- [6] MPFI group (Revol, Rouillier) INRIA <http://www.cs.utep.edu/interval-comp/interval.02/revo.pdf>
- [7] Nedialko S. Nedialkov, Vladik Kreinovich, Scott A. Starks, “Interval Arithmetic, Affine Arithmetic, Taylor Series Methods: Why, What Next?” (2003) <http://citeseer.ist.psu.edu/nedialkov03interval.html>
- [8] G. W. Walster, “Sun Forte Fortran,” <http://developers.sun.com/prodtech/cc/products/archive/whitepapers/tech-interval-final.pdf>