Computation with the Extended Rational Numbers and an Application to Interval Arithmetic

Richard J. Fateman
Computer Science Division, EECS Dep't
University of California at Berkeley
and Tak W. Yan
Computer Science Department
Stanford University

Abstract

Programming languages such as Common Lisp, and virtually every computer algebra system (CAS), support exact arbitrary-precision integer arithmetic as well as exact rational number computation. Several CAS include interval arithmetic directly, but not in the extended form indicated here. We explain why changes to the usual rational number system to include infinity and "not-a-number" may be useful, especially to support robust interval computation. We describe techniques for implementing these changes.

1 Introduction

It is well known that any rational number can be represented as an ordered pair of integers, namely numerator and denominator. In order to make this representation canonical, we usually impose the additional constraints that the greatest common divisor of numerator and denominator be 1, and that the denominator be positive.

The exact rational operations of addition, subtraction, multiplication, and division, plus a variety of other useful operations (comparison, reading, writing) form a useful and often-programmed suite of routines in the construction of systems for "symbolic and algebraic manipulation". The construction of these programs depends on the availability of arbitrary-precision integer arithmetic.

In an attempt to head off yet more duplicative programming and the introduction of incompatible names for such operations, the standard for Common Lisp [9] has established data types and names for operations for exact rational numbers. However, not all the details have been specified, and the loose ends may be tied up in several ways.

There are two rationales for extending the rational numbers, aesthetic and utilitarian. Aesthetically, the rationals form an algebraic field, and this provides a comfortable set of properties for computation. However, a computer system must respond in some way to every rational field operation including "division by 0." Simply halting computation seems unpleasant and unnecessary. Our proposal for responding to such an operation does not quite constitute a different algebraic closure of the number system, but it provides a kind of topological closure. We add a symbol for the special case of rational infinity, (the otherwise unused 1/0), and also represent "not-a-number" (NaN) (via 0/0) denoting an object that is not a member of the rational numbers. We also define appropriate extended meanings to all the field operations (addition, multiplication, division) as well as input/output. Since such a change to a representation tends to propagate effects into many parts of a system, the aesthetic judgment has to be weighed carefully after all these pieces are thought through beyond the field operations. In particular, any extension must make sense considering that the rational numbers are embedded in the reals, which are themselves embedded in the complex numbers.

The second rationale is utility. Floating-point number systems are in most ways less aesthetically pleasing than the real number system (or even the rationals) yet their utility for computer applications is well recognized. If the extension makes applications easier to write or more likely to be correct, then it may be worthwhile.

We argue that extension of the rationals in a few ways produces substantially increased utility at minimal cost. In this system we can easily simulate a model where computation is simply halted by (say) division by zero, so no utility is lost. Our hope is that our proposal can be presented as a workable and consistent prescription for a language that now includes rationals (in particular, Common Lisp), but does not yet offer much guidance as to the consequences of division by zero.

2 Extensions

We must design a treatment for the cases of division of a (zero or non-zero) quantity by zero. One approach is to check for this case, give an error message, and retreat. Except for the work described below, all systems for rational arithmetic we are aware of assume that division by zero is a non-continuable error.

We prefer to continue the calculation attempting to maintain some information as to how we left the field of rationals, with a hope that we may rejoin the rationals by some suitable further computation. (To be sure, we must also raise a flag or, at a user option, execute additional code, but the computation should normally proceed.) A simple example of this situation is the computation of $a + b/(c + 1/d)$ which is defined for $d = 0$ as a even though a division by zero has occurred (see [5] for more on this topic).

We describe two systems that in effect retain a small but useful amount of information to indicate how we departed

Submitted to 1994 International Symposium on Symbolic and Algebraic Computation, Oxford (UK), July, 1994

from the rational domain. One system allows us to compute with two additional "extended rational numbers" 0/0 and 1/0. The second system has a total of four additional symbols: the two above and symbols for negative infinity (-1/0) and negative zero (0/-1). Each of these extended systems has a number of useful properties:

1. It is essentially cost-free in implementation, even compared to giving an error message and quitting.
2. It preserves all properties of the normal rational numbers when the extended numbers are not used.
3. It allows some computations involving 0/0 or 1/0 to proceed, correctly, to a useful and justifiable answer (e.g. 1/1 divided by 1/0 is 0/1). This is a natural computation in the context of continued fractions.
4. It will produce an answer of the form 0/0 or 1/0 only when a more traditional system would have given a division by zero error at an earlier stage.
5. The IEEE floating-point standard arithmetic system provides a model for the treatment of infinities and "not-a-numbers." Systems with floating-point support may have had to deal with elements analogous to 0/0 or 1/0 and may therefore have made some key decisions in that context that can apply to rationals as well. In case these decisions have been left in abeyance, the models here may provide some impetus to make some progress there, too.

The disadvantage of these extensions is that some well-known theorems which apply to the field of rational numbers fail to hold with respect to 0/0 and 1/0. The extent to which these violations constitute hazards is fairly limited [3]. There are two exceptional circumstances in cancellation:
If $(a \cdot x)/(b \cdot x) \neq a/b$ then $(a \cdot x)/(b \cdot x)$ is 0/0.
If $(a - x) - (b - x) \neq a - b$ then $(a - x) - (b - x)$ is 0/0.
There are two exceptional circumstances in distribution:
If $a \cdot x + b \cdot x \neq (a + b) \cdot x$ then $a \cdot x + b \cdot x$ is 0/0.
If $a/x + b/x \neq (a + b)/x$ then $a/x + b/x$ is 0/0.

A rational expression computed in different ways can have at most two values in this system. It can have two values only if one of them is 0/0.

3 Models

There are at least two useful models for dealing with infinities: *projective* or *affine*. The projective model identifies +1/0 and -1/0 as a single infinity. (One way of looking at this is to consider that the "sign" of the denominator 0 is unknown, and thus so is the "sign" of the infinity). The affine model, by contrast, has signed infinities, but as a consequence also contains signed zeros. As an experiment, we have programmed both models and find that the affine model seems to require a small amount of additional checking in the algorithms for consistency. The extra cost is borne by the normal arithmetic routines, rather than in the error-handling routines, as is the case for the projective model. Yet the affine arithmetic model appears to be somewhat more useful for the application we cite later in this paper. We use affine numbers for the endpoints of intervals in implementing a more complete model of *interval* arithmetic.

There is larger context: it is appropriate to examine how each of the models makes sense in a real or complex model. For example, appropriate treatment of domains of definition of functions in the complex plane ([3]) must be considered. The Riemann sphere provides one completion for the complex numbers — the real projective model is then a slice through that sphere. By contrast, the complex affine model might be thought of as a plane with infinities of different arguments in all directions. The real Affine model is then the real-line in that plane. There are, however, other models for complex closure: plausible models include closure in the form of a cylinder or a torus.

At the moment, our implementation is such that only one model can be used at a time—which one is controlled by loading an appropriate library at run-time. Some thought has to be given to the ways in which a user might change at run-time form one model to the other, or to mix them [4]. Merging the two models into one is feasible too, at least in a language able to carry extra information on types.

3.1 The projective model

As described below, this model corresponds roughly to one that was proposed but then dropped as an option in the IEEE 754 standard model for binary floating-point arithmetic. Although the alternative, namely affine mode, appears to be the one to be adopted for the standard, the projective model is perhaps more appropriate for exact rational arithmetic intervals *per se.* The virtue of the projective reals $\mathbf{R}$ is that any *rational* function $f(x)$ of one real variable must be a continuous differentiable map of $\mathbf{R}$ to $\mathbf{R}$, despite poles, if the chordal metric is used.

From the numeric computation perspective it tends to be more conservative in the sense of giving 0/0 (not-a-number) in more circumstances than the affine mode. Kahan [4] provides more discussion on this topic. The table of operations below is more specific on this point.

In the projective model, the object 1/0 is an analog to the IEEE 754 floating-point standard's infinity. The one redeeming property of 1/0 is that its reciprocal is 0.

The model has exactly one infinity, 1/0. The form -1/0 is never produced by arithmetic operations and, if provided as input, is converted to (printed as) 1/0. In order to clarify the notation (vs. the action of division by 0) we will sometimes use ∞ as a synonym for this value.

The object 0/0 represents a canonical "not-a-number" or NaN which if operated on with any rational operation, produces 0/0. The form 0/0 should not be equated with undefined because that might mean merely "a number but unknown in value". By 0/0 we mean to provide a rational analogy to the IEEE floating-ooint standard's floating-point NaN: it is an escape notation meaning the result is not a member of this numeric type. In principle, every data type might be worth augmenting with such a reserved entity. In order to clarify the notation (vs. the action of division of 0 by 0) we will sometimes use NaN as a synonym for this value.

Neither 0 nor ∞ (reminder: we print this as 1/0) has a sign (or it is ignored), so $|0| = 0$ and $|\infty| = \infty$.

The sign of a product or quotient is "+" if the operands have the same sign, "$-$" otherwise, except if an operand is 0 or ∞ or Nan in which cases signs are absent. $-0 = 0 = 1/\infty$, $-\infty = \infty$.

$x - y$ produces the same result as $x + (-y)$.

$x - x = 0$ unless x is ∞ or NaN.

Any rational operation with at least one NaN produces NaN, and the following operations create NaN results: 0/0, ∞/∞, $\infty \pm \infty$, $0 \cdot \infty$, $\infty \cdot 0$.

Rational operations between finite and infinite results follow the expected rules indicated by the tables below.

NaNs and ∞ do not participate in the ordering of finite rationals. In particular, all of the relations $y > z$ $y \geq z$, $y \leq z$, and $y < z$ are false if y or z is ∞ or NaN. Although $\infty = \infty$ is true, every other case of $y = z$ is false if y or z is ∞ or NaN; and NaN $\neq$ NaN. (This description is largely quoted from Kahan [4], which also provides further discussion.)

For the binary operations + and *, we give tables for the logical cross-product, where $0 = 0/1$, $1/0$ is real infinity, and x and y are "normal" non-zero members of the rational field.

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

Table 1: Projective "+"

$*$	0	1/0	0/0	x
0	0	0/0	0/0	0
1/0	0/0	1/0	0/0	1/0
0/0	0/0	0/0	0/0	0/0
y	0	1/0	0/0	xy

Table 2: Projective "*"

The results of binary - and / can be deduced by the identities $a - b = a + (-b)$; $a/b = a \times (1/b)$; and the tables for unary - and 1/.

$-$	0	1/0	0/0	x
	0	1/0	0/0	$-x$

Table 3: Projective Unary "-"

1/	0	1/0	0/0	x
	1/0	0	0/0	$1/x$

Table 4: Projective "1/"

In the case of comparisons, we have to distinguish between "set-element" equality and numeric equality. As a set element, any symbol must be equal to itself. This is Common Lisp's EQ predicate. However, we propose that the the symbol 0/0 not compare "numerically equal" (Lisp's EQL) to itself, because any two occurrences of 0/0 may have totally different provenances, even assuming that some meaning can be assigned in some limiting case. We must distinguish set-equality from numerical equality for the extended rationals. EQL is therefore not even an equivalence relation because it is not reflexive! (Recall that an equivalence relation is reflexive, symmetric, and transitive.)

Similarly, if all predicates but $\neq$ return false on comparisons involving 0/0 and other elements, the comparisons do not satisfy the "law of trichotomy" — in fact we do not have a linearly ordered domain with 0/0 and 1/0 in it.

Since the projective model does not distinguish between -1/0 and 1/0, this preserves the identity 1/(1/x) = x. However, this means 1/0 + 1/0 = 0/0, since the sign of 1/0 is undetermined. Comparisons involving 1/0 must take this into account. The otherwise apparently useful arithmetic rule 1/0 + 1/0 = 1/0 which might be available if 1/0 and -1/0 were signed infinities, is sacrificed for safety.

Comparisons are discussed in the IEEE floating-point committee working group notes IEEE P754/82-2.19, in a reply by W. Kahan to W. Buchholz, (unpublished). The conclusion is that attempts to remove these ostensible comparison inconsistencies can only make matters worse. This does not interfere with the usual field operations on usual field elements, in any case.

A note on implementation: we extend the notion of gcd to include non-positive integers and zero; $gcd(a, b) = gcd(|a|, |b|)$ and $gcd(x, 0) = gcd(0, x) = x$. This extension need not impact the gcd algorithm adversely — in fact, most implementations we have encountered are already so extended. Then the usual programs for addition, multiplication and division of rational numbers, represented as pairs of integers, hold for the operations which include 1/0 and 0/0. For example, to compute $a/b + c/d$, compute $r := (ad + cb)$, $s := bd$, and $g := gcd(r, s)$. The only change necessary is to test: (if $g = 0$ return 0/0 else) return $(r/g)/(s/g)$.[1] It is not necessary to examine any of the numerators a, c or r (for example, to see if they are 0).

3.2 The affine model

In the affine model, we have both positive infinity 1/0 and negative infinity $(-1/0)$. Consequently we are forced to have two zeroes: positive 0 (represented by 0/1) and negative (-0) (represented by 0/-1). Among other reasons for preferring it, this model is more suited to transcendental functions [6] allowing one to describe branch cuts more specifically.

The tables below give the addition, multiplication, negation, and reciprocation for this model. Note that $+0 = -0$ arithmetically, but division by them always produces unequal results. Other ways of distinguishing them (e.g. by `copysign` [2]) should be provided. We choose to define the result of $(+0)+(-0)$ to be +0. This is not always the correct choice, but the complications that would ensue by allowing yet a third zero (unsigned 0) are unappealing.

Kahan [4] provides the following rules for the affine extension:

$-0 = +0$ arithmetically, although they can be distinguished in other ways.

The sign of a product or quotient is "+" if the operands have the same sign, "$-$" otherwise, regardless of the operands being finite, zero, or infinite.

The sign of a sum of terms with the same sign matches it, regardless of the operands being finite, zero, or infinite.

$x - y$ produces the same result as $x + (-y)$ but not the same as $-(y - x)$ when this is -0 because ... $x-$ produces +0 (arbitrarily chosen) for every finite x.

Any rational operation with at least one NaN produces NaN, and the following operations create NaN results: 0/0, ∞/∞, $\infty \pm \infty$, $0 \cdot \infty$, $\infty \cdot 0$.

Rational operations between finite and infinite results follow the expected rules indicated by the tables below.

[1]More elaborate programs are sometimes employed to multiply polynomial fractions. Extracting the gcd of b and d first can sometimes result in a more efficient calculation, since the cost of two smaller polynomials gcd's can be lower than the cost of a single gcd computation with larger inputs. The use of two gcd computations probably has a lower or non-existent payoff in the integer fraction case, since integer arithmetic, even the arbitrary-precision version which is used here, is fairly fast.

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 5: 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 6: Affine "*"

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

Table 7: Affine Unary "-"

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

Table 8: Affine "1/"

Implementing these operations require modest checking for special cases.

4 Traps, Pre-Substitution

It should ordinarily be possible to use functions which will, on production of 0/0, cause a trap to a user-defined function rather than just continuing. Either special version of the functions may be used, or the same functions — with a check against a flag of some sort — can be used. The user-defined function, according to a recommendation of W. Kahan, could be constructed so as to

- use a value provided in anticipation of the event (so-called pre-substitution) by which one could redefine 0/0 in some circumstance), or
- change the course of computation.

In Lisp, the logical existing mechanism to use is is analogous to `catch` and `throw`, implemented through the `error` system. An example of such a technique is given in Appendix I.

5 Interval Arithmetic

A useful application of extended rational numbers is in interval arithmetic [3], [11], [8], [7], [10]. The model for interval arithmetic we use comprises both projective and affine real models: the projective model is implemented for intervals themselves, although an affine model is used for the endpoints.

Let r, s, p, and q denote any extended affine rational numbers. If $r < s$, then $[r,s]$ is the *interior* interval $\{x \in \mathbf{R} : r \leq x \leq s\}$ and $[s,r]$ is the *exterior* interval $\{x \in \mathbf{R} : x \leq r$ or $s \leq x\}$. In accordance with Kahan's suggestions [3], allowances can be made for intervals which, if an endpoint is ± 0 or $\pm 1/0$ indicate whether or not to exclude that endpoint by a sign. The complications of having open/closed endpoints at *arbitrary* points seems not worth the bother, however. Here is the pattern:

$[+0, 1/0]$ includes projective positive reals except (projective, hence unsigned) 0 and (projective, hence unsigned) 1/0.
$[-0, 1/0]$ includes 0 but excludes 1/0.
$[+0, -1/0]$ includes 1/0 but excludes 0.
$[-0, -1/0]$ includes 0 and 1/0.

The corresponding exterior intervals with reversed endpoints can be deduced by intersection with the projective reals. For example,
$[1/0, +0]$ includes projective negative reals and also includes 0 and 1/0.
For doubly-infinite intervals, consider

$[-1/0, 1/0]$ includes all finite reals, but excludes 1/0.
$[1/0, -1/0]$ includes all finite reals as well as 1/0.
$[+0, -0]$ includes all *non-zero* finite reals as well as 1/0.

We can represent any finite non-zero real r uniquely by $[r, r]$. We can choose to represent 0 by $[-0, +0]$ or $[+0, +0]$ and 1/0 by $[1/0, 1/0]$, leaving an otherwise unused interval representation, $[-1/0, -1/0]$ to be used for the "empty interval".

5.1 Addition

If either of the endpoints of either of the two intervals is 0/0, then the sum is "indeterminate" or $[0/0, 0/0]$. Otherwise, the sum of two intervals $[r,s]$ and $[p,q]$ is $[r+p,\ s+q]$ if $(r \leq s \ \&\ p \leq q)$, or $(r \leq s \ \&\ p > q \ \&\ r+p > s+q)$ or $(r > s \ \&\ p \leq q \ \&\ r+p > s+q)$. The sum is $[-1/0, 1/0]$ in all the other cases.

5.2 Subtraction

The negation of $[r, s]$ is $-[r, s] = [-s, -r]$, and $[r, s] - [p, q] = [r, s] + [-q, -p]$.

5.3 Multiplication

For multiplication, we find the product by applying the following rules *in order*.

1. If either of the endpoints of either of the two intervals is 0/0, the product is [0/0,0/0].
2. If one of the intervals is empty, the product is the empty interval.
3. If one of the intervals is $[0, 0]$ and the other includes an endpoint that is $1/0$ or $-1/0$, the product is [0/0,0/0]; otherwise it is $[0, 0]$.

4. If one of the intervals is $[-1/0, 1/0]$, the product is $[-1/0, 1/0]$.

5. If one of the intervals is exterior (with normal endpoints), say $[b, a]$ with $a < b$, we split it into two intervals $[-1/0, a]$ and $[b, 1/0]$. The product is the union of the products of these two intervals with the other multiplicand. The union is of intervals that are necessarily contiguous (an exterior interval). The product of two exterior intervals is handled similarly.

6. Finally, if all the above fail, cross-multiply the endpoints of the intervals to get four extended rational numbers; let *MAX* and *MIN* be the maximum and minimum. The product is [*MIN*, *MAX*]. (This can be economized if endpoints of one interval have like signs).

5.4 Division

The reciprocal of $[r,s]$ is $1/[r,s] = [1/s, 1/r]$, and $[p, q]/[r, s] = [p, q] \cdot (1/[r, s])$.

6 Additional Notes

Programs (written in Common Lisp) for all the operations specified here, plus considerable additional detail, are available from the authors. They could be used to augment any Common Lisp [9] standard system to provide computation over the extended rational numbers or this projective model of (interior and exterior interval arithmetic with affine rational or IEEE floating-point endpoints.)

Unfortunately, as general and extensible as is the Common Lisp design, adding new "numeric" types cannot be done entirely smoothly — the user does not have access to the type hierarchy directly. Details are presented in the program documentation.

7 Utility and Conclusions

The evidence to date on the usefulness of rational interval arithmetic is still sparse. The difficulties are two-fold. Firstly, even using rational operations, the tendency is for the sizes of numerators and denominators in the endpoints to grow exponentially with the number of operations. In the case of large intervals, it seems implausible that, for the most part, highly precise endpoints are justifiable. Another operation of conservatively "abbreviating" an interval to represent a bound – but with "brief" endpoints — should be provided. Rounding endpoints down and up are required.

Secondly, the need for non-rational operations (log, sin, square-root) occurs fairly often. Although some of these program have also been written for use with these representations, there is another parameter needed to determine how precisely the interval endpoints should be located for exact inputs.

Given these problems with the usefulness of the system, it is apparent that intervals combined with an arbitrary-precision floating-point system may very well be more attractive for general use. Implementing such a system requires careful attention to rounding up and down as appropriate to keep any points in the interval range of a function from wandering – even slightly – outside the interval domain. It is rather easy to do this sloppily and somewhat miss the point of interval arithmetic, as was done in the first version of Mathematica [12].

Nevertheless, exactness for performing arithmetic intervals should continue to be of use in strict error bound calculations.

8 Acknowledgments

Thanks to especially to W. Kahan for numerous discussions. J. Coonen supplied reference material on Apple's SANE [1] standard, as well as general and comments. Guy Steele, Jr. and Steven Haflich, members of the Common Lisp standardization committee have also been helpful.

Work on this topic has been, in years past, sponsored by the US Dept. of Energy, Army Research Office, and National Science Foundation, most recently NSF Infrastructure Grant number CDA-8722788 and NSF Grant number CCR-9214963.

References

[1] Apple Computer Inc. *Apple Numerics Manual, 2nd ed.* Addison Wesley Publ. 1988.

[2] IEEE Computer Society Microprocessor Standards Committee Task P754, a standard for binary floating-point arithmetic, (see, for example, draft 8.0, Computer 14, 3 Mar. 1981, 52-63)

[3] W. Kahan. A More Complete Interval Arithmetic. Unpublished lecture notes for a summer course at the Univ. of Michigan, Ann Arbor, 1968.,

[4] W. Kahan. What Numbers?, class notes for course Math 128A, Fall, 1992, Univ. of Calif. Berkeley.

[5] W. Kahan. Presubstitution and Continued Fractions, EECS Dep't, Univ. of Calif. Berkeley unpublished, 1987. See also, Rational Arithmetic in Floating-Point, PAM-343, Ctr Pure and Appl. Math. Univ. of Calif, Berkeley, 1986.

[6] W. Kahan. Branch Cuts for Complex Elementary Functions. in *The State of the Art of Numerical Analysis*, A. Iserles and M. J. D. Powell (eds.) Oxford Univ. Press, 1987.

[7] Ramon E. Moore. *Methods and Applications of Interval Analysis.* SIAM studies in applied mathematics, 1979 also, *Reliability in computing: the role of interval methods in scientific computing*,Academic Press, 1988.

[8] K. Nickel (ed). Interval Mathematics 1985, Proceedings of the Int'l Symp. Freiburg i. Br., FRG, September, 1985, Lect. NOtes in Comp. Sci, 212, Springer-Verlag, 1985.

[9] Guy L. Steele, Jr. *Common Lisp the Language, 2nd ed.*, Digital Press, 1990.

[10] Ulrich W. Kulisch, Willard L. Miranker. *Computer Arithmetic in Theory and Practice*, Academic Press, 1981.

[11] Jean Vuillemin, Exact Real Computer Arithmetic with Continued Fractions. *IEEE Trans. on Cmptrs. 39* 1990, 1087–1105.

[12] S. Wolfram. *Mathematica—A System for Doing Mathematics*, 2nd edition, Addison-Wesley, 1991.

Appendix I — Affine Rationals

In this section we describe the specific features of the extended rational (affine) model as we have implemented it in Common Lisp (CL).

This package includes operations and representations for rationals and extended rationals (`erats`) including Infinities (1/0, -1/0) Not-a-number (0/0) and signed zeros 0, -0 (= 0/-1).

The options of trapping on divide by zero or invalid are provided (separately).

The "signed zero" means that the denominator of an `erat` might be negative, but then the denominator is in fact -1 and the numerator is 0.

Two global variables are used for enabling traps. These are `*rat-dbyz-trap-enable*` and `*rat-invalid-trap-enable*` The user should use these two global variables only via programs similar to those illustrated below as `rat-dbyz-catch` or `rat-invalid-catch`. That is in part why we made the global variable names long.

Also, two global variables can be queried by the user to see if an un-trapped 0/0 was produced in a sequence of operations. These are `*rat-dbyz-flag*` and `rat-invalid-flag*` These will be set to `T` if such a value has been produced. Hence, arrays of `erats` need not be scanned for 0/0.

Finally, four global variables are used:

`*rat-zbyz-value*`, `*rat-ztimesinf-value*`, `*rat-infminusinf-value*`, and `*rat-infbyinf-value*`.

These can be used for presubstitution [5] for values 0/0, $0 \cdot \infty$, $\infty - \infty$, and ∞/∞, respectively, as illustrated by the program `rat-zbyz-presubs` below.

The extended rationals and operations on them are defined using the Common Lisp Object System (CLOS). Operations (methods) on `erat` objects are defined to combine them with other numeric objects, print them, and convert them.

A printing method is set up to print an `erat` with a decimal point to the left of the "/" to distinguish it from a rational number. Thus we see 1./2 instead of 1/2. This artifact would be removed if the extended system were actually adopted for Common Lisp.

The usual user-access to the number representation is the function `create-erat` which is the top-level function to make a new `erat`. `(create-erat x y)` makes the number x/y. It assumes nothing about the inputs x and y except that they are either `erat` types or CL real numbers (we define the type `realnum` to be `number` (as defined in CL) but not `complex`). The values of x and y can also be erats. If the second argument y is missing, it is taken to be 1.

Because the CL system does not provide good hooks to change the operation of built-in numeric functions like "+", we spell-out the names for our operations. The functions provided include `plus, times, quotient, reciprocal`. Each of these is programmed so that if the objects being combined do not look like numbers or `erats`, a "quoted form" is returned. This looks like, for example, `(Plus x 3)`. Additional functions include `negate, numer` (access the numerator), `denom` (access the denominator), and `absol` for absolute value.

Comparison function are `greaterp, lessp, erat=`.

Using IEEE floating-point rules, here are some cases that might be puzzling:

Comparison	Result
$0/1 > 0/-1$	false
$any > 0/0$	false
$0/0 > any$	false
$1/0 = 1/0$	true
$-1/0 = -1/0$	true
$-1/0 = 1/0$	false
$0/0 = any$	false
$0/1 = 0/-1$	true

Actually IEEE 754 has 26 functionally distinct comparisons composed from the setting of 4 relation flags (greater, less, equal, unordered) and an exception-enabling flag. We have provided only 3 of the comparisons, plus a separate flag for operations on unordered. There are some subtleties because not-equal is not the same as less-than-or-greater-than. The latter gives an exception for unordered.

Although it is plausible to do so, we have not provided a number of other items that would complete the "integration" of `erats` into CL. We'd rather provoke some discussion at this point. We've left unprogrammed for the moment: the `expt` function; coercion to and from other formats; `difference, remainder, zerop, plusp, minusp, notequal`, etc. (which can be defined by analogy with "normal" rationals.

We can use floating-point NaNs for results of `(coerce inf-rat 'float)`, `(coerce inf-rat 'single-float)`, `(coerce inf-rat 'double-float)` and `(coerce inf-rat 'long-float)`.

Coercion of a `nan-rat` to a float type produces `nans` of that type. In Franz Inc's Allegro CL on machines with IEEE-754 compatible arithmetic, we have the forms

`#.excl::*nan-single*`, `#.excl::*infinity-single*` `#.excl::*negative-infinity-single*`, `#.excl::*nan-double*`, `#.excl::*infinity-double*`, and `#.excl::*negative-infinity-double*`.

Full semantics for the float-nans is not specified in the proposed CL standard. It may be, in fact, something we should specify in the course of refining this package. Although IEEE 754 defines invalid, underflow, overflow, divide-by-zero and inexact, for our rational model only divide-by-zero and invalid can even happen. (It should be pointed out that prior to this standard, computer manufacturers including DEC, Cray, and CDC provided some sort of reserved or special operands. The IEEE standard did not "invent" this concept; it just goes much further in specifying potentially useful operations on such operands. Neither the standard, nor our proposal here, forces a system to continue computation if the programmer prefers a simple "halt.")

To be explicit, we give some examples of how the principal subtlety of this package can be used — the trap handling.

The function `rat-dbyz-catch` is an example of a program which can be used to interact with the exception handling. In this case, `(rat-dbyz-catch e val)` evaluates the expression `e` and in case there is a divide by zero, returns the value represented by `val`. If you wish to write a program segment such that the occurrence of a divide-by-zero in `(f x)` will abandon the computation and just return 1, use `(rat-dbyz-catch (f x) 1)` instead of just `(f x)`.

```
(defmacro rat-dbyz-catch (e val)
  `(let* ((*rat-dbyz-trap-enable* t)
          (res (catch 'rat-dbyz ,e)))
     (if (eq res 'rat-dbyz) ,val res)))
```

This next example shows how to catch "invalid."

```
(defmacro rat-invalid-catch (e val)
  `(let* ((*rat-invalid-trap-enable* t)
          (res (catch 'rat-invalid ,e)))
     (if (eq res 'rat-invalid) ,val res)))
```

An alternative treatment known as presubstitution [5] would set up the system so that if certain exceptional operations are executed, normal numbers are returned instead. For example, if you wish to specify that all divisions of 0 by 0 in the computation of `(f x)` should be treated as 1, `(rat-zbyz-presubs (f x) 1)` will do precisely that, where we define:

```
(defmacro rat-zbyz-presubs (e val)
  `(let ((*rat-invalid-trap-enable* nil)
         (*rat-zbyz-value* 1))
      ,e))
```

The code for the affine model of rationals is rather straightforward, and could be read by anyone moderately familiar with Common Lisp and seriously interested in the model. The code could be made considerably more complicated, continuing in the spirit of the IEEE 754 standard. Other possible complications include implementing the other 23 comparisons or a "compare and branch;" `remainder`, `difference`, `copysign`, `finite`, `isnan`, `unordered`, `class`.

Although we are using the IEEE 754 model, not every issue resolved there is relevant. For example extra traps, namely inexact, over/underflow, cannot occur. Directed rounding modes are irrelevant; square-root is not closed in the rationals, conversion of binary to/from decimal is done by CL; denormalized numbers and their consequences as well as `scalb`, `logb`, `nextafter`, are all unnecessary.

Code is available on request from the first author, fateman@cs.berkeley.edu.