Exact Real Arithmetic: A Case Study in Higher Order Programming

Hans-J. Boehm
Robert Cartwright
Mark Riggle

Rice University

Michael J. O'Donnell

University of Chicago

Abstract

Two methods for implementing *exact* real arithmetic are explored One method is based on formulating real numbers as functions that map rational tolerances to rational approximations. This approach, which was developed by constructive mathematicians as a concrete formalization of the real numbers, has lead to a surprisingly successful implementation. The second method formulates real numbers as potentially infinite sequences of digits, evaluated on demand. This approach has frequently been advocated by proponents of lazy functional languages in the computer science community Ironically, it leads to much less satisfactory implementations. We discuss the theoretical problems involved in both methods, give algorithms for the basic arithmetic operations, and give an empirical comparison of the two techniques. We conclude with some general observations about the lazy evaluation paradigm and its implementation.

1. Introduction

Although many theoretical computer scientists are familiar with the notion of the *representable* or *constructive* real numbers, there has been surprisingly little research on the subject of performing exact real arithmetic on computers. Engineers and scientists have traditionally solved computational problems involving real numbers by using a subset of the rational numbers to approximate real numbers and limited-precision arithmetic to approximate true arithmetic. Since physical measurements have only finite accuracy, calculated values that depend on measured data are inherently inexact. Consequently, limited-precision arithmetic has a strong physical justification — assuming that the round-off errors introduced by limited-precision arithmetic are comparable in size to the errors introduced by the inaccuracy of the original data. During the last thirty years, numerical analysts have devised a wide variety of clever methods for computing important functions that are relatively insensitive to "round-off" error. Given the success of this enterprise, it is not surprising that existing programming languages completely ignore the possibility of performing exact real arithmetic

This research was partially supported by NSF grant number DCR84-03530.

In this paper, we explore the feasibility of performing exact real arithmetic on a computer with an emphasis on implementations that employ lazy evaluation. As we explain in more detail in Section 3, lazy evaluation has been suggested as an important optimization in the implementation of higher order data like the real numbers because it adroitly avoids recalculating digits that were already calculated at an earlier point in a computation. The lazy implementation of the representable reals is deeper and more interesting subject than might be expected. In particular, the computability of basic arithmetic operations such as addition critically depends on the choice of data representation.

The obvious representations based on positional radix notation and continued fractions are formally inadequate. Fortunately there are some more sophisticated representations with all the necessary mathematical properties, but even these more sophisticated representations are plagued by performance problems in practice.

We contrast this approach with one based on the constructive mathematicians' definition of real numbers (cf [Bis 67]), which we argue does lead to a reasonably efficient implementation of exact real arithmetic

1.1. Motivation

In the $\mathbf{R}^n$ project at Rice University, we became interested in the subject of exact real arithmetic because we had an immediate practical need for it. As part of the $\mathbf{R}^n$ Fortran programming environment, we are building a program validation system that tests programs against executable formal specifications. The program validation system is a pragmatic alternative to a formal program verification system. Instead of trying to prove that a program satisfies its formal specifications, the program validation system evaluates the program and its specifications on a representative collection of test data [Cart 81] In essence, the validation system tests the the *verification* *conditions* for the program rather than proving them. We call this process *formal program testing*.

We are interested in formally specifying numerical programs in a very high level executable specification language called TTL (Tentatively Typed LISP) that roughly resembles a lexically scoped dialect of LISP (*e.g.* SCHEME) augmented by a comprehensive constructive data type definition facility [Cart 80,82a,83] that accommodates lazy data constructors. Since the natural way to specify the behavior of a numerical algorithm is to relate the calculated answer to the true answer[1], to serve as a specification language for numerical software, TTL must accommodate the real numbers and standard arithmetic operations (addition, subtraction, multiplication, division, exponentiation, comparison etc.) on them. Since TTL expressions must be executable, there is no alternative but to support exact real arithmetic in TTL.

Although the design and implementation of the $\mathbf{R}^n$ programming environment is the catalyst that prompted us to study exact real arithmetic, we believe that a reasonably efficient implementation of the representable reals is a potentially useful tool to help numerical analysts, scientists, and engineers solve computational problems where they either need very high precision or want to bypass the myriad of programming details (*e.g.* scaling, order of evaluation, and error estimates) required in conventional numerical programming The representable reals provide a more abstract, mathematically elegant framework for performing computations where efficiency is not a primary concern In some cases (*e.g.* Gaussian elimination with iterative refinement) the precision advantages can be maintained even if only a small fraction of the necessary calculations are carried out using representable real numbers; floating point arithmetic may still be used for the remainder.

Although any computation involving the representable reals — no matter how efficient the underlying implementation — will almost certainly require more computational resources than the corresponding computation using conventional floating point arithmetic, the plummeting cost of these resources suggests that exact arithmetic may become an attractive option in some scientific and engineering applications. Moreover, since many of the atomic steps in exact arithmetic operations (*e.g.* operations on individual digits) can be performed in parallel, parallel architectures should further improve the situation. In the future, we anticipate that exact real arithmetic will be a standard feature in very high level programming languages intended for program prototyping and effective specification — just as floating point arithmetic is a standard feature in general purpose high-level languages.

[1]An alternative is to write specifications based on *backward error analysis* [Wil 65]. In this case, program assertions relate the perturbed problem solved exactly by the calculated answer to the original problem. This approach imposes the same computational requirements on the specification language as the approach (corresponding to *forward error analysis*) discussed above.

2. Formal Foundations

Since there are uncountably many real numbers but only countably many possible data representations, we obviously cannot represent every real number in a computer Fortunately, there is a countable subset of the real numbers, appropriately dubbed the *constructive reals* [Bis 67], *recursive reals*, or the *representable reals* that is closed under all the functions that normally arise in analysis. Moreover, all of the basic operations $\{+, -, *, /\}$ are computable on the representable reals — assuming that the numbers are expressed in a suitable representation

The intuition underlying the definition of the representable reals is simple: for the representation of a real number x to have computational significance, there must be a procedure for computing finite approximations to x to arbitrary accuracy. More formally, a real number r is *representable* if and only if there exists a computable function f on the rational numbers with the following property. for any given rational tolerance δ, $|f(\delta) - r| \leq \delta$. Any such function f is called a *functional representation for* x Two functional representations f and g are equivalent, denoted $f \equiv g$ iff they represent the same real number.

A function F on the representable reals is *computable* iff there exists a corresponding computable functional F', such that for any functional representation f of a real number x, $F'(f)$ is a representation for $F(x)$. In particular, F' must yield equivalent results if applied to equivalent arguments. This definition applies whether or not the range of F also consists of the representable reals. Of course many functions on functional representations do not correspond to functions on the representable reals because they do not respect the equivalence relation $\equiv$.

Although it may seem pedantic, the distinction between a representable real number and its representations is important for two reasons First, as we mentioned earlier, the *computability of the basic operations hinges on the choice of representation*. Second, as we discuss in section 5, there are difficulties in limiting the available operations on representable reals to those that are completely independent of representation.

Although all of the basic arithmetic operations on representable real numbers are computable, the standard comparison operation $<$ is not. It is easy to prove this fact using a simple reduction argument that reduces the halting problem to whether or not a particular function represents a real number less than a given constant. In practice, we can partially avoid this problem by either tolerating a partial $<$ operation that diverges if the two operands are equal, or equivalently[2] by defining a quasirelational comparison operator $<_\epsilon$ that is accurate to the specified rational tolerance ϵ. Given any rational number ϵ, $<_\epsilon$ has the following definition:

[2]Some facility for execution interleaving is necessary to define $<_\epsilon$ in terms of the partial $<$.

$$x <_\epsilon y \quad = \quad \begin{cases} \textit{true} \text{ if } x < y-\epsilon \\ \textit{false} \text{ if } x > y+\epsilon \\ \text{either } \textit{true} \text{ or } \textit{false} \text{ otherwise} \end{cases}$$

Such an operation is total, but ***does not*** correspond to a computable function on the representable reals, because its behavior depends on the particular representation for inputs that lie within the specified tolerance ϵ.

3. Alternate Representations

Although it is clearly possible to construct an implementation of the representable reals based purely on functional representations, this representation does not appear well-suited to general computations in which an iterative or recursive flow of control may repeatedly force the reevaluation of particular numbers to differing accuracies.[8] A pure functional implementation must recalculate the requested rational approximation from scratch every time it is invoked.

Any implementation will clearly benefit from some facility for remembering the most precise previously calculated approximation One possibility is to simply extend the representation to include the most precise known approximation along with the function to compute arbitrary approximations. Another more drastic, and particularly elegant, approach is to use a lazy infinite sequence of digits to represent a real number In this scheme, subsequent requests for more accurate approximations simply extend the evaluated portion of the sequence.

Although the lazy representation of the representable real numbers appears natural and elegant, it is surprisingly difficult to devise a satisfactory formulation The classical mathematical literature ([see Knu 69]) describes two fundamentally different systems of notation for the real numbers that can be implemented as lazy data structures: the positional radix model and the continued fraction model In naive form, the positional radix model defines a representable real as an infinite lazy sequence consisting of a leading unbounded integer specifying the signed whole (integral) part of the number followed by a sequence of bounded digits $(f_1, f_2, ...)$ specifying the fractional part $0.f_1 f_2 \cdots$ of the number. The continued fraction representation has a similar format: it is a potentially infinite lazy sequence consisting of a signed unbounded integer specifying the whole part of the number followed by a potentially infinite sequence of unbounded positive integers $(d_1, d_2, ...)$ specifying the ***denominator terms*** in the continued fraction representation

$$\cfrac{1}{d_1 + \cfrac{1}{d_2 + \cdots}}$$

of the fractional part of the number. The continued fraction is finite iff the represented number is rational.

Unfortunately, there is a subtle flaw in both of these formulations. The evaluation mechanism underlying lazy sequences is monotonic in the sense used in domain theory [Sco 81]: Once a digit has been computed it never changes. (In the case of positional radix notation, this implies monotonicity with respect to the numeric ordering.) Although this property is what makes lazy representations so attractive from the standpoint of efficiency, it makes the standard arithmetic operations uncomputable in some cases, as the following example demonstrates.

Consider the following situation in the naive positional radix model with radix 10. Let x and y denote the repeating decimal numbers .4444 ... and .5555 . respectively. The sum $x+y$ has the value .9999 ... = 1.0 but the addition operation cannot determine whether the units digit in the fraction should be 0 or 1 — no matter how many digits of x and y it inspects. The unknown contents of their infinite tails could force the units digit to be either a 0 or 1. If the sum of the unexplored tail of x plus the unexplored tail of y generates a carry then the units digit in the result must be 1. On the other hand, if the sum of the unexplored tails is less than an infinite sequence of 9's, then the units digit must be 0. In cases like this one, the addition operation must collect an infinite amount of information before it can generate the first digit in the answer.

The preceding example is not an isolated phenomenon. Exactly the same problem arises in many different contexts in both the radix expansion model and the continued fraction model It is easily shown formally, for example, by reduction from the halting problem, that neither addition, subtraction, multiplication, nor division is computable using these naive representations

In general, for any reasonable lazy sequence representation of a nontrivial subset of the reals and some representable number x, there must be a representation ρ for x such that every prefix of ρ can be extended to a representation of each number in an open interval surrounding x. This is false for the decimal or continued fraction representation of 1. A more formal statement and proof is given in the appendix.

The situation for the continued fraction scheme is, in some sense, even worse. The principal theoretical advantage of this representation scheme is its most significant computational liability. In regular continued fraction notation, every real number has a unique representation; moreover, the unique representation of every rational number is a finite sequence. If we require that we be able to detect finiteness of the representation, this immediately leads to a contradiction. The calculation of $\sqrt{2} * \sqrt{2}$ must clearly generate an infinite sequence as its result.

Fortunately, there is a simple generalization of the positional radix model that overcomes these problems We can allow the range of digits to be larger than the base,

[8] A related problem arises even with simple nested multiplications. In a practical implementation it is necessary to obtain upper bounds on the operands in order to bound the necessary precision required to obtain the result. Depending on the details of the representation this may require 1 or more trial evaluations of the operand to low precision.

making the notation highly redundant. It is most convenient to use a variant of balanced radix notation [Knu 69], in which we simply allow digits to be negative as well as positive. Thus the decimal number 1.1 could also be represented as 2 (-9).

Representational redundancy enables the evaluator to generate carries (up to a bound determined by the degree of redundancy) for a digit that has already been generated and store them in the generating digit position by using redundant digit values (Similar techniques have also been widely used to minimize carry propagation in arithmetic hardware)

4. Algorithms

We have developed reasonably efficient algorithms for computing all of the standard arithmetic and quasirelational operations for a modified form of the functional representation. We have also developed corresponding algorithms using the balanced redundant positional radix notation The latter are described in detail below. They are basically the standard arithmetic algorithms taught in grammar school, adapted to produce digits in left to right order

4.1. Addition and Subtraction

We first give the addition algorithm for the functional representation. In this modified functional representation, a real number x is represented by a program that maps each (usually negative) integer n into an integer m such that

$$(m-1)\,b^n < x < (m+1)\,b^n$$

where b is a given positive base >1. Informally an integer n is mapped into a scaled approximation of x, accurate to $\pm b^n$. The modified functional representation, which is derived from a representation proposed by Bishop [Bis 67], is computationally more convenient than the original one presented in Section 3

To add two real numbers x and y, represented by functions x' and y' , we must construct a representation z' such that for all n

$$(z'(n)-1)b^n \;<\; x+y \;<\; (z'(n)-1)b^n$$

To produce such a representation it suffices to return a function that evaluates x and y to one more digit of precision than that required of the result, and then returns an appropriately scaled version of the sum.

```
fn_rr_add ≡
    lambda (x' , y' )
        lambda (n)  (x'(n-1) + x'(n-1))/b
```

where integer division is rounded.

Here we have assumed $b \geq 4$. To argue informally, that the result function correctly represents $x + y$, it suffices to observe that the approximations of x and y each contribute at most $\frac{1}{b}$ to the error in the result produced by the representation of $x + y$. The rounding involved in the division introduces another possible error of at most $\frac{1}{2}$. Thus the total possible error is less than 1.

We define a lazy representation as a pair $[e, f]$ consisting of of an (immediately evaluated) unbounded integer exponent e, together with a lazy sequence f of digits in a redundant balanced radix representation scheme where the maximum digit value *maxdig* is the additive inverse of the minimum digit value *mindig* and the base b is greater than *maxdig* but less than 2*maxdig* The lazy sequence representing the mantissa is always a fraction, with an implied decimal point on the left. Thus $[e, f]$ denotes

$$b^e \sum_{i=1}^{\infty} b^{-i} f_i$$

Although it is not difficult to devise algorithms for minimally redundant representations (where $b = 2maxdig$), these algorithms are less efficient because they require one more position of look-ahead in the inputs Some form of partial normalization is desirable in practice, but omitted here for the sake of simplicity. We use the lazy constructor (*.*) to form lazy lists. This constructor is identical to LISP *cons* except that it is lazy in its second argument. We use pattern-matching notation to denote the selectors corresponding to pairing [*,*] and the stream constructor (*.*). Matching a pattern variable against the tail of a lazy sequence does *not* force evaluation of the tail For example,

let (x0.xtl) ≡ f

binds $x0$ to the first element and binds *xtl* to the delayed tail of f, which is forced only when *xtl* is evaluated (It is enlightening to convince oneself that this is a necessary constraint)

The following addition routine *lazy_add* adds two unnormalized lazy real numbers and produces an unnormalized result. The meat of the algorithm is embedded in the routine *ff_add_wc*, which takes two lazy sequences denoting fractional mantissas as inputs and produces lazy sequence of digits consisting of the carry digit in the sum followed by the fractional part of the sum

```
lazy_add ≡
lambda ([xe,xf], [ye,yf])
    if xe = ye then
        [xe + 1, ff_add_wc(xf,yf)]
    else if xe > ye then
        [xe + 1, ff_add_wc(xf,shift(yf,xe-ye)]
    else
        [ye + 1, ff_add_wc(shift(xf,ye-xe),yf)]


shift ≡
        lambda (f,n);
            if n = 0 then
                f
            else
                (0 . shift(f,n-1))
```

```
{ Add two lazy digit sequences representing    }
{ fractions. The leading digit of the result   }
{ sequence corresponds to the generated carry. }
ff_add_wc ≡
lambda ((x0.xtl),(y0.ytl));
  let sum ≡ x0 + y0
      [carry,z0] ≡
          if (sum ≥ maxdig) then
              [1, sum - b]
                { Compensate for anticipated carry }
          else if (sum ≤ mindig) then
              [-1, sum + b]
                { Compensate for anticipated borrow }
          else [0, sum]
      { sum = carry * b + z0 }
      { mindig < z0 < maxdig }
  in
      (carry . df_add( z0,ff_add_wc(xtl, ytl)))
```

```
{ Add a single digit to the first digit of a fraction, }
{ assuming no carry can be generated.                  }
df_add ≡
        lambda (d,(f0.ftl));
            ((d + f0) . ftl)
```

The algorithm works by summing two digits to produce a carry and a remainder where the carry cannot be affected by a subsequent carry from the next digit position into the remainder position.

This algorithm directly generalizes to a polyadic addition that accommodates up to $(maxdig-mindig)-b$ operands. To accommodate more operands the algorithm must look more than one digit ahead in the inputs.

Negation algorithms are trivial in either system of representation In the lazy scheme we simply complement each digit. In the functional scheme we produce a new function that returns the negation of the value returned by the original function. For future reference, we give the algorithm for negating fractional mantissas represented as a lazy sequence of digits:

```
ff_minus ≡
        lambda((x0.xtl));
            (-x0 . ff_minus(xtl))
```

4.2. Multiplication

Multiplication is a little more complicated than addition, but still manageable. A scaled functional implementation of multiplication is similar to the implementation of addition, with the added complication of precomputing an a priori bound on the arguments in order to determine the precision needed for the arguments. Bishop simply uses the integer part plus 1. Our implementation uses a more precise algorithm, which is not presented here. It is worth noting that in either case the arguments must be evaluated more than once, usually with differing precisions. (In some other representations, such our lazy sequence representation, the exponent can be used to give a pessimistic bound.)

The following lazy multiplication algorithm assumes $mindig = -b+1$ and $maxdig = b-1$. This serves to simplify the presentation

```
lazy_mult ≡
lambda([xe,xf], [ye,yf]); [xe+ye, ff_mult(xf,yf)]
```

```
ff_mult ≡
lambda ((x0.xtl), (y0.ytl));
{ Decompose the product into four partial products. }
{ z0 and z1 represent the product of the two        }
{ leading digits.                                   }
  let z0 ≡ x0*y0 div b
      z1 ≡ x0*y0 rem b
  in
      ff_add( (z0 . (z1 . ff_mult(xtl,ytl)),
          ff_add_wc( df_mult(x0,ytl), df_mult(y0,xtl) ) )
```

```
{ Multiply a single digit, with an implied decimal }
{ point on the left, by a fraction, obtaining      }
{ another fraction.                                }
df_mult ≡
lambda(d,(f0.(f1.ftl)));
  let z2 ≡ d*f1 rem b
      z01 ≡ d*f0 + d*f1 div b
      z1 ≡ z01 rem b
      z0 ≡ z01 div b
        { (f0*b+f1)*d = z0*b^2 + z1*b + z2        }
        { Carry from remainder of expansion may   }
        { affect z1 by ±1.                        }
        { Absval(z01) is bounded by (b-1)b.       }
        { Thus either absval(z0) < b-1 or         }
        { absval(z1) < b-1                        }
      [adj_z0, adj_z1] ≡ if z1 = b-1 then
                             [z0+1, z1-b]
                         else if z1 = -(b-1) then
                             [z0-1, z1+b]
                         else
                             [z0, z1]
  in
      (adj_z0 . df_add(adj_z1,
                    df_add_wc(z2, df_mult(d,ftl)) ) )
```

```
{ The routine ff_add again assumes no carry }
{ can be generated                          }
ff_add ≡
  lambda((x0.xtl),(y0.ytl));
      df_add(x0 + y0, ff_add_wc(xtl,ytl))
```

```
df_add_wc ≡
lambda(d, (f0.ftl))
   let sum ≡ d + f0
      [carry, z0] ≡ if (sum ≥ b-1) then
                        [1, sum-b]
                    else if (sum ≤ -b+1) then
                        [-1, sum+b]
                    else
                        0
   in
      (carry . (z0  ftl))
```

4.3. Division

There are a number of alternatives for division, both for the functional and lazy approaches. In the functional approach, it is possible to proceed as before, and effectively perform a division on large integers Since this is relatively expensive on conventional architectures, a competitive alternative is to approximate the inverse iteratively using Newton's method, until the desired precision is reached This is particularly attractive if a previous approximation is available as a starting point.

The same alternatives exist for the lazy sequence approach. Here we present the central algorithm involved in the direct (not Newton's method) approach As discussed below, this is not the most efficient algorithm

The algorithm below takes two mantissas as arguments, and produces the mantissa of the quotient. We restrict our attention to the case $b \geq 16$, and continue to assume $maxdig = -mindig = b-1$. We also assume that the dividend and the divisor have been scaled so that the dividend is smaller than the divisor, and the leading digit of the divisor is at least $\frac{b}{2}$. (This may require multiplying both dividend and divisor by a single digit, as suggested in [Knu 69]. The normalization of the divisor may diverge if it is 0, but then division must inherently diverge in this case.)

```
{ Return (x - (digit/b) * y)*b.           }
{ It is known that abs(result) < 1        }
remainder ≡
lambda(x, y, digit)
   let
      (r0.rtl) ≡ ff_add(x, ff_minus(df_mult(d,y)))
   in
      if r0 = 0 then
         rtl
      else if r0 = 1 then
         df_add(b, rtl)
      else {r0 = -1}
         df_add(-b,rtl)
```

```
{ Return a fraction corresponding to x/y.  }
{ The operands x and y                      }
{ are scaled as described above.            }
ff_div ≡
   lambda(x, y);
      let
         (x0.(x1.xtl)) ≡ x,
         (y0.(y1.ytl)) ≡ y;
         x01 ≡ x0*b² + x1*b;
         y01 ≡ y0*b + y1;
         q0 ≡ round(x01/y01)
      in
         (q . ff_div(remainder(x, y, q), y))
      ni
```

The heart of the correctness proof is to show that the recursive call to ff_div satisfies the normalization assumptions on the arguments, and in particular to show that

$$remainder(x,y,q) < y$$

This can be expanded as follows

$$\left|(x-(\frac{q}{b})y)b\right| < |y|$$

$$\left|bx-(y)\,round(\frac{x01}{y01})\right| < |y|$$

We need to argue that $round(\frac{x01}{y01})$ approximates $\frac{bx}{y}$ with sufficient accuracy .

The quantity $y01$ approximates b^2y with an absolute error of at most 1, and thus, since $y \geq \frac{7}{16}$, by a relative error of at most $\frac{16}{7}b^{-2}$. Thus $\frac{b^2}{y01}$ approximates $\frac{1}{y}$ with a relative error of less than $3b^{-2}$ Since $\frac{x01}{y01} < b$, the absolute error introduced by approximating y by its two leading digits is at most $\frac{3}{b}$ and thus $\frac{3}{16}$ Since $x01$ approximates b^3x to within b, this approximation introduces an absolute error into the quotient approximation of at most $(\frac{1}{b})(\frac{1}{|y|})$ or $(\frac{1}{b})(\frac{16}{7})$ or at most $\frac{1}{7}$. Rounding introduces an absolute error of at most $\frac{1}{2}$ Thus

$$\left|b\frac{x}{y}-round(\frac{x01}{y01})\right| < \frac{3}{16}+\frac{1}{7}+\frac{1}{2} < 1$$

$$\left|bx-(y)round(\frac{x01}{y01})\right| < |y|$$

4.4. Discussion

In comparing the algorithms for the functional and sequence approaches, it becomes apparent that each has weaknesses. As was pointed out above, the functional approach, at least in its simple form, is prone to excessive reevaluation of expressions.

The lazy sequence approach discussed above suffers from a less obvious, but in practice more crucial, performance problem. We would like to take advantage of machine arithmetic for the operations on digits In order to minimize the number of machine arithmetic instructions involved, each digit should be approximately the size of a machine word. But consider an expression such as

$$x_1 + (x_2 + (x_3 + (\cdots + x_n)))$$

Assume we are only interested in the integral part of the result. The repeated application of the binary addition algorithm will result in the evaluation of x_i to i digits beyond the decimal point. Thus x_n will be evaluated to n fractional digits, and thus typically $16n$ or $32n$ fractional bits, when it is clear that at most n (and really only $\log n$) fractional bits are necessary.

This problem is particularly severe for the division algorithm presented above The calculation of the n^{th} digit of the quotient effectively requires the $n+1^{st}$ digit of a remainder produced by an expression of the form

$$dividend - q_1 b^{-1} divisor - \cdots - q_{n-1} b^{-n+1} divisor$$

Since the result of the above expression is required to n digits, and the dividend is nested inside $n-1$ subtractions, by the above argument, it will be evaluated to $2n$ digits. Thus the evaluation of the expression

$$x/3/3/3/3/ \ldots /3 \ (m \text{ divisions})$$

to a single digit will force the evaluation of x to 2^m digits. Similar problems occur with the divisor They can of course be at least partially solved by more clever and less elegant coding. For example, the division algorithm could take an unevaluated sum (and a shift count) as its first argument, so that the remainder expression abovecould be evaluated right to left

The basic problem with lazy representations is that of the granularity they require in precision specifications. Each approximation must be accurate to an integral number of digits. When arithmetic operations are performed, they must discard information about the inputs, because they cannot present the most accurate answer. After an addition operation has computed the arguments to $\pm b^{-k}$ it can only produce an answer with a known accuracy of $\pm b^{1-k}$ in spite of the fact that enough information is available to generate the answer to a tolerance of $2b^k$.

In the functional scheme we can avoid the problem by making the precision argument as precise as possible, subject to the practical constraint that calculations on precision arguments should remain inexpensive In our version we are still limited to a tolerance that is a power of the base But there is no efficiency penalty for using a small base

5. Representable Reals as an Abstract Data Type?

It is tempting to provide the user of representable reals with a small collection of functions, such as the arithmetic operations, and to let him/her use these to construct other computable functions that might be needed Ideally this would be done in such a way that the representation is completely hidden.

We already encountered one problem with this approach — a total comparison operation must inherently be approximate, and furthermore its exact behavior must depend on the representation. In fact, it is a corollary of Rice's Theorem from computability theory that any computable, total, boolean valued function on the representable reals must be constant.

We take a program to be sequential if it diverges whenever it attempts a divergent subcomputation. Now consider a sequential program P that computes a function from representable real numbers to representable real numbers Assume that P is only allowed to apply total computable (and thus non-representation-revealing) first-order[4] functions from a finite set F to representable real arguments None of the if- or while- conditions in P may depend on the input to P. Thus P's control flow can't depend on its input. Thus P must either diverge, or be equivalent to a fixed composition of the functions in F

An easy diagonal argument shows that for each such set F of total functions, there is a computable total function on representable reals that is not computed by any program P of the above form.

The preceding argument can be strengthened to allow partial functions in F by observing that if a boolean expression in P depends on P's input, then the value of that boolean expression cannot be a total function of the input parameter Thus, if P is purely sequential, it cannot compute a total function. In particular, it can still not compute the diagonal function from the preceding argûment.

It thus appears difficult to model the representable reals as an abstract data type without representation-dependent functions.

6. Practical Experience

Our first implementation is based on the modified functional approach using scaled approximations and is written in Russell [Boe 85a,85b] [Dem 85]. A number x is represented primarily as a function mapping an unbounded integer n to an unbounded integer m such that

$$(m-1)b^n < x < (m+1)b^n$$

The implementation keeps track of the best known approximation to a number in addition to the function, so that reevaluation can be minimized. Multiplicative inverses are computed directly for small precisions, with Newton's method used to refine results to higher precisions. In the

[4]Parameters and results to functions in F should be representable real numbers or integers, Boolean values etc. We disallow functions as parameters or results.

latter case, the best previous approximation is used as the starting point. A square root function is implemented using a similar approach. The exponential function and natural logarithm are implemented using Taylor series approximations, with some initial scaling to hasten convergence Several desk calculator style interfaces have been built for this implementation.

In addition, we have several partial implementations based on the lazy sequence of digits view Below we discuss two of these. They are both direct translations of the above algorithms, one into Scheme, and one into Russell. The Russell implementation includes the four basic arithmetic operations. The Scheme version currently lacks division

These efforts have so far been less successful. Table 1 presents a timing comparison for multiplication based on the following benchmark program·

```
{ In the following, x and third are constructive reals, }
{ i is an integer.                                      }
x := 1;
third := 1/3;
      { For lazy sequence versions, third is directly }
      { bound to a list of 3s.                        }
      { In the functional case, the division is       }
      { performed once.                               }
for i := 1 to 5 do
   x := x * third;
      { Constructive real multiplication }
for i := 1 to 5 do
   x := x * 3;
      { Constructive real multiplication }
print x;
```

The lazy sequence implementations used base 10 with a -9 to 9 digit range. The choice of base 10 is clearly not optimal. We adjusted for this by only requiring 10 digits to be printed for the sequence based implementations, but requiring either 50 or 100 decimal digits to be printed by the functional version. (The timings for the functional version include base conversion overhead, the sequence timings do not.)

All timing results refer to a VAX 11/750 running UNIX[5]. All measurements were made repeatedly, alternating between different implementations No significant variations were observed Mean measurements are reported here.

The Scheme version was compiled using Kent Dybvig's Chez Scheme compiler (version 1 1), the fastest implementation available to us For the Scheme version, program load time was subtracted from the total execution time The measurements are probably somewhat biased against Scheme for several reasons. First, Russell is statically typed, thus making the code generator's job easier Second, lazy lists are a built-in data type in Russell. Thus their implementation was carefully hand-coded. Third, one of the authors is very familiar with the internals of the Russell compiler, and was thus probably more successful at tuning the constructive reals implementation for Russell than for Scheme.

The times reported are user-mode cpu seconds (System-mode times were comparable for the lazy sequence implementations, but with high random variations They were also much less for the functional version) The Scheme version used a *maximum* of about 600 Kbytes *real* memory, the Russell sequence version used about 1 2 Mbytes, and the functional implementation used about 180 Kbytes[6]. All implementations based on Russell preallocate 1 Megabyte of *virtual* memory, which is then expanded as necessary

The last 2 lines of table 1 refer to versions that print the result to the indicated precision after each multiplication This forces incremental evaluation to increasing precision, and should thus demonstrate the advantage of the lazy sequence approach. This kind of incremental evaluation would presumably not add significantly to the times for lazy sequences.

The last line refers to execution times for the package with a desk calculator interface. Multiplications by one third were replaced by divisions by 3 (which are considerably more expensive). The arithmetic operations were typed in from the keyboard. Each loop was (manually) executed 10 rather than 5 times

These results have lead us to the following conclusions regarding the functional implementation:

1 Execution speed is acceptable for this application and, by extrapolation, for some of the applications mentioned in the introduction

2 There appears to be much more overhead involved in bookkeeping than in performing the necessary operations on huge integers During the multiplication and division experiment described in the last line of figure 1, the processor only spent about 18 percent of its time performing large integer multiplications and divisions. In this case, scaling high precision results to lower precision values is also fairly significant, about 9 percent of the time was spent performing more than 1900 shift operations on integers. As expected, this does change for extremely high precision calculations A calculation of exp(1) to 1000 digits (which takes about 160 cpu

Lazy sequences (Scheme)	66 4
Lazy sequences (Russell)	30 3
Functional (Russell, 50 decimal digits)	1 2
Functional (Russell, 100 decimal digits)	1 5
Functional (Russell, 50 digs., w/printing)	2 2
Functional calc., 20 iter , 70 digs	about 20

Table 1

[5]Trademarks of Digital Equipment Corporation and AT&T Bell Laboratories, respectively.

[6]These figures were measured by the Berkeley UNIX (C Shell) time and ps commands.

seconds) spends 80 percent of its time multiplying integers.

3. In a number of cases, more complicated analysis of the necessary operations could have led to significant performance improvement. For example, a long sequence of additions results in a demand for unnecessary precision on the first argument. An explicit, and much less elegant, representation of real numbers as expression trees could eliminate this problem with a preliminary tree balancing phase.

As is apparent from figure 1, the lazy sequence implementations currently show less promise. This is not as surprising as it seems at first In our benchmark, the first multiplication is eventually carried out to 20 more digits than the final one. The total number of procedure calls (excl. garbage collector and run-time system) exceeds 90,000. Virtually all of these return closures, and thus require heap allocated activation records.

Our experiences in building the lazy sequence based implementations can be summarized as follows.

1 This approach to implementing the constructive reals appears not to be competitive given existing programming language implementations on conventional hardware.

2. Many existing compilers were not designed to support higher order data and lazy evaluation efficiently. In spite of this, it appears unlikely that an improvement in compiler technology alone would drastically alter our comparison. The Russell based implementation executes approximately one user level procedure call every 330μsecs. Each call typically requires the heap allocation of a closure. (Currently the majority of calls allocate at least an activation record, a cons node, and a closure) It is very unlikely that any compiler would be able to reduce the average execution time of such a procedure to 60 μsecs on the given machine. (One invocation of a C language recursive Fibonacci function requires about 35 μsecs.) It also appears extremely difficult to reduce the total number of procedure calls by more than a factor of 2 through procedure integration. Thus a factor of 10 improvement in execution time through better compilation techniques seems remotely possible, but unlikely. Yet even this would not result in a competitive implementation.

3. It is relatively easy to convince oneself of the partial correctness of a program based on lazy lists. The only subtlety is introduced by the fact that free assignable variables appearing in lazily evaluated expressions refer to their value at time of (delayed) evaluation. Thus lazy evaluation in imperative languages cannot simply be viewed as delaying evaluation to save time.

On the other hand, even in purely applicative languages, termination is a very subtle issue. (For example, ***ff_add_wc*** diverges if ***df_add*** forces evaluation of ***ftl*** to a cons cell with a closure representing the tail.) Resource requirements are even more subtle. We know of no good technique for reasoning about space requirements of the lazy algorithms

As mentioned above, we found support for lazy data structures to be marginal in some implementations which claimed to support them. Code optimization appears even more critical in this context than in massive floating point programs. The following two problems appear common in existing compilers:

1 If programming with higher order objects is to be encouraged, closures must be kept small. A lazy list should not take up significantly more space than the prefix that has been evaluated. At least one conventional LISP implementation[7] fails badly in this respect, most probably by failing to collect certain activation records. This also argues for copying partial environments into closures as in [Card 84], rather than building closures that point to activation records The latter approach does not make it clear to the garbage collector that a given closure relies only on a small number of bindings, and not the complete environment at the point at which it was formed. This can make irrelevant data appear accessible and thus uncollectible

2. It is important to keep hardware arithmetic functions efficiently accessible. This is frequently not the case due to tag bits used by the run-time system to facilitate garbage collection and perhaps dynamic type checking.

7. Related Work

The only alternative to limited precision arithmetic that has received much attention from computer researchers is exact arithmetic on rational numbers. Every rational number can obviously be represented by two unbounded integers (*e.g.*, a list consisting of two LISP ***bignums***). The principal advantage of rational arithmetic is that it is possible to implement the fundamental arithmetic operations {+, -, *, /} exactly, avoiding round-off errors. Unfortunately, rational arithmetic has two serious disadvantages that make it unsuitable for many of the potential applications of exact real arithmetic First, it does not accommodate computations involving the use of functions that map rational inputs into irrational results (such as exponential and trigonometric operations). Second, rational arithmetic is unsuitable for long calculations because it typically produces rational numbers with huge numerators and denominators. The advantage of representable real arithmetic is that the actual amount of computation is driven by the required precision of the result, whereas rational arithmetic always computes full precision results, whether or not they are needed.

The theory of constructive real numbers has been explored by a number of mathematicians [Bis 67] [Bri 79] [Myh 72] Their emphasis has been on formal foundations

[7] *Not* Chez Scheme.

rather than efficient computation. Some of the issues involved have also been explored by recursion theorists (cf [Rog 67])

A few unpublished papers [PJ 84] [Cart 82b] contain fragmentary discussions of proposed lazy representation of the representable reals using the naive implementations discussed in Section 4 The most interesting account is an unpublished MIT technical note[8] in which Bill Gosper proposes a lazy evaluation scheme before any of the seminal papers on the subject were published. Gosper recognizes that his algorithms do not work for all inputs and suggests implementing a hybrid system that uses approximate arithmetic as a last resort

The computability issues arising with the sequence of digits view were explored by [Myh 72]. The basic problems involved in this view were apparently well know to recursion theorists before then (cf [Rog 67], [Wie 80]). Myhill also discusses (and dismisses) another approach based on "located" real numbers

The use of redundant notation to limit carry propagation is well known among hardware designers In a slightly disguised from, it forms the basis of fast hardware multiplication algorithms Its uses in variable precision arithmetic were explored by [Avi 61], [Avi 64], [Atk 75], [Owe 79]

The use of redundant notation in the context of constructive real arithmetic was suggested in [Wie 80], in an unpublished paper by Carl Pixley[9], and in [O'Do 85]. It was independently rediscovered (once more) by the other authors of this paper. [O'Do 85] gives an equational version of the addition algorithm.

8. Future Work

Although we feel that we have an adequate implementation of representable real numbers, there is clearly a lot of potential for improvement in performance. Some of the possible sources of improvement are trivial, for example the use of hardware floating point when its precision is adequate, the use of faster integer multiplication algorithms for huge integer calculations, etc.

There are at least two possible ways to obtain more substantial performance improvements First, the algorithms used in our implementations could probably benefit from further tuning. Secondly, as mentioned above, there appears to be room for improvement in the implementation of the underlying programming languages, particularly in the treatment of closures and machine arithmetic.

It is an open question whether there is any way to build a competitive implementation based on lazy sequences. In an attempt to eliminate the granularity of precision problem discussed in section 4.4 we are exploring variations on the lazy formulation of the real numbers. These variations technically violate the monotonicity condition that prohibits a digit from changing once it has been computed. In particular, the last digit in an approximation may change, as long as the preceding digits remain fixed. We call this property *weak* monotonicity Instead of forcing the precision of an approximation to be an integral number of digits, the precision of the last digit is explicitly stored, along with the closure needed to generate more accurate approximations. As long as the sum of the absolute value of the last digit and the tolerance is less than the base b, more accurate approximations can be obtained simply by changing the last digit and appending additional digits (including another tolerance for the last digit). This makes it possible to avoid discarding information in arithmetic operations. Unfortunately, the cost is added complexity in the the algorithms for arithmetic operations.

It is also conceivable that the lazy sequence algorithms given here might become practical with some substantial support from the underlying hardware (*e.g* in somehow allowing small digit sizes without penalty).

We are only starting to explore the applications of representable real arithmetic For testing purposes, it would be useful to run standard numerical programs using exact arithmetic to produce exact results. It is clear that this is not always possible without additional information from the programmer, particularly for iterative algorithms. On the other hand, it seems possible in at least some cases The real question is how much difficulty is introduced by the lack of a traditional comparison operator.

Such a facility for executing numerical programs exactly might also give other useful information to the programmer. It would be easy to keep track of the number of digits needed at each point in the computation to derive the result with the desired accuracy. This should make it easier to determine how much floating point precision is needed at various points in the program.

9. Acknowledgements

We would like to thank Mike Fagan and Jerry Fowler for their contributions to an earlier draft of this paper, and for some preliminary implementations of lazy sequence based constructive reals. We would also like to thank Venita De Souza for implementing most of the lazy sequence based algorithms discussed above Her implementations were used to obtain the timing comparisons presented above The members of the program committee provided us with many helpful comments.

[8]Gosper, Bill, "Continued Fraction Arithmetic", HAKMEM *Item 101B, MIT AI Memo 239, Feb 1972 (?).*

[9]"Demand Driven Arithmetic", around 1984.

References

[Avi 61] Avizienis, A., "Signed-Digit Number Representations for Fast Parallel Arithmetic", Institute of Radio Engineers Transactions on Electronic Computers, 1961, p. 389.

[Avi 64] Avizienis, A , "Binary-Compatible Signed-Digit Arithmetic", AFIPS Conference Proceedings 26, 1 (1964), pp. 663-672

[Atk 75] Atkins, D. E., "Introduction to the Role of Redundancy in Computer Arithmetic, IEEE Computer 8, 6 (1975), pp. 74-76

[Bis 67] Bishop, Errett, *Foundations of Constructive Analysis*, McGraw-Hill, New York, 1967.

[Bri 79] Bridges, D. S., *Constructive Functional Analysis*, Pitman, London, 1979.

[Boe 85a] Boehm, Hans, Alan Demers, and James Donahue, "A Programmer's Introduction to Russell", Technical Report 85-16, Department of Computer Science, Rice University.

[Boe 85b] Boehm, Hans, and Alan Demers, "Implementing Russell", Technical Report 85-25, Department of Computer Science, Rice University. Was also presented at the SIGPLAN '86 Compiler Construction Conference.

[Card 84] Cardelli, Luca, "Compiling a Functional Language", Conference Record of the 1984 Symposium on LISP and Functional Programming, pp. 208-217.

[Cart 80] R. Cartwright, "A Constructive Alternative to Axiomatic Data Type Definitions", Proceedings of 1980 Symposium on LISP and Functional Programming, Stanford University, August 1980, pp. 46-55.

[Cart 81] R. Cartwright, "Formal Program Testing," Proceedings of the Eighth Annual Symposium on Principles of Programming Languages, Williamsburg, Va , January 1981, pp. 125-132

[Cart 82a] R. Cartwright and J. Donahue, "The Semantics of Lazy and Industrious Evaluation," Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Carnegie-Mellon University, August 1982.

[Cart 82b] Cartwright, Robert S , et al., "Rn: An Experimental Computer Network to Support Numerical Computation", Technical Report, Mathematical Sciences Department, Rice University, 1982.

[Cart 83] R. Cartwright, "Recursive Programs as Definitions in First Order Logic", SIAM J. Computing, May 1984

[Dem 85] Demers, Alan, and J. Donahue, "Data Types are Values", ACM Transactions on Programming Languages and Systems 7, 3 (July 1985), pp 426-445.

[Gol 83] Golub, Gene H., and Charles F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1983.

[Knu 69] Knuth, Donald E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Addison Wesley, 1969.

[Myh 72] Myhill, J., "What is a Real Number?", American Mathematical Monthly 79, 7 (1972), pp. 748-754

[O'Do 85] O'Donnell, Michael J., *Equational Logic as a Programming Language*, MIT Press, 1985

[Owe 79] Owens, R. M. and M. F. Irwin, "On-Line Algorithms for the Design of Pipeline Architectures", Annual Symposium on Computer Architecture, Philadelphia, 1979, pp 12-19

[PJ 84] Peyton Jones, Simon L., "Arbitrary Precision Arithmetic Using Continued Fractions", INDRA Note 1530, Department of Computer Science, University College London, 1984

[Rog 67] Rogers, Hartley Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967. See especially p. 371.

[Scot 81] D Scott, Lectures on a Mathematical Theory of Computation, Technical Monograph PRG-19, Oxford University Computing Laboratory, Oxford, 1981.

[Wie 80] Wiedmer, E., "Computing with Infinite Objects", Theoretical Computer Science 10 (1980), pp. 133-155

[Wil 65] Wilkinson, J H., *The Algebraic Eigenvalue Problem*, Academic Press, London, 1965

Appendix

The purpose of this section is to show that any "reasonable" implementation of exact real arithmetic that is based on lazy sequences must have what we will refer to as the "interior containment" property. Roughly stated, this means that each number x must have a representation such that any prefix determines an interval containing x, such that x is not one of the endpoints. Neither the standard decimal representation nor any obvious variant of the continued fraction representation share this property.

This theorem is trivial if we insist that there be a computable mapping from the "standard" functional representation to this lazy sequence representation. In the following we show that it is necessary even if we only insist that addition and subtraction be computable. In particular, we do not even require that the collection of numbers representable in this scheme is the same as our standard notion of the "representable reals".

A ***lazy*** representation of the real numbers assigns to some subset of the real numbers infinite computable sequences $\{a_i\}$ where $a_i \in D$. D may be infinite.

A lazy representation is said to be an ***interval representation*** if it satisfies the following constraints.

(1) All rational numbers are representable by some sequence.

(2) There is a computable function M that maps any finite sequence $\{a_i\}_{i=1}^{n}$ into (the endpoints of) a rational (open or closed) interval Φ such that the rational numbers contained in the interval are exactly those that can be obtained by extending Φ

(3) Every sequence represents a number. That is, for every sequence $\{a_i\}$ there is a number x, such that

$$\bigcap_{n=1}^{\infty} M(\{a_i\}_{i=1}^{n}) = x$$

(4) For some representable number x, there is a representation, $x = M(\{\rho_i\}_{i=1}^{\infty})$ such that every finite prefix is mapped to an interval of nonzero length. That is, for every n, there is some sequence $\{\phi_i\}_1^m$ such that $x \notin M(\{\rho_i\}_{i=1}^{n} \,||\, \{\phi_i\}_1^m)$. where " || " denotes sequence concatenation We say this x is ***densely represented***

This asserts that for some representation, the tail of the sequence is important.

A point, x is in the ***interior*** of an interval, Φ, iff there exists a rational ϵ such that an ϵ-neighborhood of x is contained in Φ.

Definition: An ***interval*** representation has the property ***interior containment*** iff for any real number x that has a representation, there is a representation, ρ, such that x is in the interior of the interval defined by every prefix of ρ

Note that this requirement will exclude the standard decimal and continued fraction representations

Definition: If an interval representation supports total computable addition and subtraction operators, it is a computable interval representation; or abbreviated as CIR.

Note that for an operation to be considered computable, it must be possible to produce an arbitrarily long prefix of the result representation from finite prefixes of the operand representations.

Theorem: Any CIR has the interior containment property

Proof: Assume we have an interval representation coding that does not have the interior containment property Then there is a representable number x such that for all representations, $\{\rho_i\}_{i=1}^{\infty} = x$, there is an n such that $M(\{\rho_i\}_{i=1}^{n})$ does not have x in its interior. We will reduce a recursively inseparable set problem to the existence of computable addition and subtraction operators on this representation. First several lemmas are necessary.

Lemma: The two sets

$$S_1 = (x,y) \mid \phi_x(x) \text{ halts before } \phi_y(y)$$

$$S_2 = (x,y) \mid \phi_y(y) \text{ halts before } \phi_x(x)$$

are recursively inseparable. Here $\phi_x(x)$ denotes the computation of the x^{th} Turing machine on input x.
Proof: Well known, *e.g.* by reduction from [Rog 67] p. 94

Lemma 1: Any CIR must have the number 0 densely represented.
Proof: Let x be densely represented. Consider the representation for $x - x$.

Lemma 2: Any CIR must have a representation, $\{\rho_i\}_{i=1}^{\infty}$, for the number 0 that has 0 in the interior of $M(\{\rho_i\}_{i=1}^{n})$ for all n.
Proof: Assume we are given a pair (x,y) Let X be 0 if $\phi_x(x)$ does not halt, and denote some $\epsilon > 0$ where $\epsilon \in M(\{\rho_i\}_{i=1}^{n})$ and $0 \notin M(\{\rho_i\}_{i=1}^{n})$ if $\phi_x(x)$ halts in n steps. Let Y be similarly defined for y. Clearly such X and Y are expressible in any CIR since 0 is densely represented. Now consider the representation for $X - Y$; it is zero if $\phi_x(x)$ and $\phi_y(y)$ do not halt, some $\epsilon > 0$ if $\phi_x(x)$ halts before $\phi_y(y)$ or some $\epsilon < 0$ if $\phi_y(y)$ halts before $\phi_x(x)$. If zero does not have a representation with the interior containment property, then some finite prefix of the representation of $X - Y$ must either exclude all positive numbers or all negative numbers.

Let S be the set of all (x,y) pairs such that positive numbers are excluded from $X - Y$. Since this can be detected, S is recursive. Since S separates S_1 from S_2 this is a contradiction.

By the lemma, a representation for 0 must exist with interior containment for 0. So consider $x + 0$, where 0 is represented as in lemma 2

Corollary: In a CIR, no number may have a decidable equality relation between all its representations.