INDRA Note 1530

INDRA Working Paper

Arbitrary precision arithmetic using continued fractions

Simon L Peyton Jones

ABSTRACT: Functional languages supporting lazy evaluation invite novel applications, where conventional languages do not provide appropriate support for the problem. In this paper we present an application of functional languages to arbitrary precision real arithmetic, using an unusual technique based on continued fractions, and depending crucially on lazy evaluation.

The precision of computer arithmetic calculations is normally decided by the programmer in advance, and is often hard to alter subsequently. Furthermore, without a formal analysis of the calculation, answers may be produced to spurious accuracy, and the computer gives no help in establishing error bounds for the result. The technique presented here performs real arithmetic with guaranteed error bounds, in which the precision of the result can be arbitrarily increased without recommencing the calculation.

Department of Computer Science

University College London

1. Introduction

Lazy evaluation, a feature provided by some functional languages, is to some extent a solution in search of a problem. It is well known that lazy evaluation allows the construction and manipulation of infinite data structures, but the new applications thus opened up are still being explored.

In this paper, we present a novel application of functional languages to arbitrary precision real arithmetic, where real values are represented by a possibly infinite sequence of terms of a **continued fraction.** These terms are only computed when required to give further accuracy to the final result, so that no unnecessary work is done. This demand-driven evaluation would be hard to implement in any other way, but fits very naturally into a lazy functional context. A further application of lazy evaluation within this system is that of performing arithmetic on infinite series.

In addition, expressing the computation in a functional form opens up the possibility of parallel evaluation of subcomputations. This area will not be explored in this paper.

Typical computer arithmetic packages offer single or double precision integer and floating point arithmetic. Using these facilities effectively requires the programmer to ensure that the range of values he wishes to represent falls within the range provided, and, more importantly, that all intermediate values will be held to sufficient precision. Errors in this latter respect are quite common, and give results which are sometimes

totally spurious, though no indication is given by the machine.

The problem is normally solved by "overkill", that is performing the calculations to a precision so great that all the intermediates will be held to sufficient precision, and then only printing the significant digits of the final result. The machine does not help in establishing the error bounds for the result, and this task has to be performed by the programmer.

The method presented here performs real arithmetic with a guaranteed error bound for the result, and allows this error bound to be arbitrarily decreased until it is acceptably small, without recommencing the calculation. Furthermore, the technique allows approximate quantities to be specified (eg 6.34 +- 0.04), in which case the error bound on the result will appropriately reflect this uncertainty. Calculations involving only rational numbers will be computed exactly, when sufficient precision is demanded (this is untrue of floating point systems, in which many rationals have recurring expansions). No unnecessary work is done (as is the case with multiprecision arithmetic), and no information is discarded (as with roundoff and truncation).

The method is based on continued fractions, whose elegant properties have long been known (see for example [Hard38] or [Khin64]). However, the classical material does not seem to address the question of performing arithmetic on continued fractions. The only work that has been done on continued fraction arithmetic, to the author's knowledge, is that of Gosper ([Gosp80] and [Gosp81]), whose ideas are seminal to this paper.

The idea of performing arithmetic using infinite objects has also been addressed in [Wied80], though from a more theoretical point of view.

A primitive package based on this method has been written in Sasl, a functional language, and a more sophisticated one is under construction. The primitive package is very short, and is included as an appendix. This is believed to be the first time that lazy evaluation has been brought together with continued fraction arithmetic.

2. Continued fractions

In this section, we give a basic introduction to continued fractions, beginning with an example. The continued fraction for 2.31 is

$$2 + \cfrac{1}{3 + \cfrac{1}{4 + \cfrac{1}{2 + \cfrac{1}{3}}}}$$

This may be more compactly written [2, 3, 4, 2, 3]. We may regard this as saying that 2.31 is

```
about 2
  but not really 2, rather 2 + 1/3
    but the denominator is not really 3, rather 3 + 1/4
       but the denominator is not really 4, rather 4 + 1/2
         but the denominator is not really 2, rather 2 + 1/3
```

A general continued fraction is of the form

```
a(0) + b(0)/(a(1) + b(1)/(a(2) + b(2)/( ... )))
```

and it may be finite or infinite. A **regular** continued fraction is one whose b terms are all equal to 1, and whose a terms are all integers greater than or equal to 1 (except a(Ø), which may be Ø or negative). Regular continued fractions can be denoted [a(Ø), a(1), a(2), ...].

Regular continued fractions have the following desirable properties:

(i) There is a unique regular continued fraction for each real number. Even infinity has a continued fraction, [], the empty one.

(ii) All rationals have finite continued fraction expansions.

(iii) We may truncate a continued fraction f = [a(Ø), a(1), a(2), ...] by taking a finite initial segment [a(Ø), a(1), ... a(n)]. Then the rational, s(n), whose expansion is [a(Ø), a(1), ... a(n)] is the closest rational to f with such a small denominator. These rationals, are called the **approximants** of f, and form a sequence of better and better rational approximations to f, each of which is "best" in the sense just defined.

(iv) The approximants s(n) = p(n)/q(n) can be generated by a pair of recurrence equations thus:

```
p(n) = a(n)*p(n-1) + p(n-2)
q(n) = a(n)*q(n-1) + q(n-2)
```

These approximants are automatically in their lowest terms.

All these properties are in the classical literature, and of themselves suggest that continued fractions might be a good number representation for computers. In particular, in a lazy functional language, a continued fraction may be represented by a lazy list, or stream, which may be infinite or finite. This list may be produced by one function, and consumed by another in a demand-driven way. Terms of the continued fraction will only

be produced when the consumer needs the extra precision which they provide. Periodic continued fractions can be represented by lists with loops in them, and even non-periodic infinite fractions can often be generated by simple functions (e, for instance).

However, all this is only useful if we can perform arithmetic on continued fractions; fortunately this can be done by a method due to Gosper [Gosp80, Gosp81].

3. Continued fraction arithmetic (Gosper)

The challenge is to produce functions which will peform arithmetic operations on continued fractions, producing a continued fraction as their output, in such a way that the terms of the operands are consumed only when necessary to produce terms of the result, so that result terms begin to appear as early as possible.

The method for the four arithmetic operations (+, -, *, /) is little known, and so will be presented here. The trick is to calculate not x+y, x*y etc, but rather

$$z(x,y) = \frac{axy + bx + cy + d}{exy + fx + gy + h}$$

or (a b c d)/(e f g h) for short

where a..h are integers. This function clearly specialises to give the four operations by suitable choice of a..h.

Now, x is a continued fraction, [x(0), x(1), ...] say. Hence

$$x = x(0) + 1/x'$$

where

$$x' = [x(1), x(2), \ldots]$$

So we can rewrite z as a function of x':

$$z(x',y) = \frac{a(x(0) + 1/x')y + b(x(0) + 1/x') + cy + d}{e(x(0) + 1/x')y + f(x(0) + 1/x') + gy + h}$$

(#) $\qquad = (ax(0)+c \;\; bx(0)+d \;\; a \;\; b)/(ex(0)+g \;\; fx(0)+h \;\; e \;\; f)$

Note that this operation preserves the basic form of z. This operation corresponds to ingesting a term of x, and expressing the dependency of the result on the rest of x, viz x'. Since the value of x can vary between x(0) and x(0)+1 as x' varies between 1 and infinity (we know that all the x(i) are in this range), we would expect z to vary over a range which reflects this. We might say that z "knows that x is in the range x(0) to x(0)+1".

The range of variation of z as we vary x' between 1 and infinity thus gives us some knowledge of the value of the result. We can narrow this range by ingesting another term of x, since

$$x' = x(1) + 1/x''$$

where

$$x'' = [x(2), x(3), \ldots]$$

This will incorporate knowledge about the range of x' into z (ie it varies between x(1) and x(1)+1).

Since z is **linear in x**, we know that z(x',y) must vary between z(1,y) and z(infinity,y) as x' varies between 1 and infinity; it

is always inside this range if the denominator does not change sign, and always outside it if the denominator does change sign. Thus we can establish the range of variation of z by considering its value at the extreme values of x.

z is a function of two arguments, x and y, and the above scheme applies equally well to y, so we can compute the entire possible range of z over all variations in x and y by taking the union of the four intervals:

```
        (zll, zlh), (zlh, zhh), (zhh, zhl), (zhl, zll)
where
        zll = z(l,l)               zlh = z(l,infinity)
        zhl = z(infinity,l)        zhh = z(infinity, infinity)
```

Now suppose that the integer part of z, which we will write {z}, is constant over its entire range. Then

```
        z = {z} + l/z'
```

where z' > l. Thus the first term of the continued fraction for z is {z}. Let us egest this term, and find z'.

```
                    l
        z' = ---------
              z - {z}

(*)          = (e  f  g  h) / (a-{z}e  b-{z}f  c-{z}g  d-{z}h)
```

So the operation of egesting a term of z also preserves the basic form of z; indeed z was chosen precisely to be preserved over ingestion and egestion.

Summary

In summary, the method works in the following way. The

multiplication (for example) program consumes terms of x and y, and produces terms of the product x*y. Its state consists of the values of the 8 integers a..h. The program examines the range of z over the extremes of x and y, and if z's integer part is constant, it outputs a term of the result, giving a new state defined by (*). If the range of z is too large, it narrows the range by ingesting a term of x or y, using (#) to compute the new state.

There are various strategies for choosing which of x or y to ingest; for example, choose the one which causes the widest variation in z.

4. Evaluation and enhancements of the method

The method we have described fulfills the requirements stated at the beginning of the last section. In this section, we discuss some of the shortcomings of the method, and suggest possible solutions.

4.1 Control of precision

One drawback of the method is that the increase in accuracy of the result with each successive term is dependent on the particular numbers involved. For instance, if the result of a calculation is 2.0001, it may well be produced as [2,1000], and all the accuracy comes at once. On the other hand, a continued fraction for 2.31 is [2,3,4,2,3], which supplies its increasing precision in smaller doses. Since the result may only be required to 2 decimal places, say, the extra precision in the first example represents wasted work.

The reason for this "lumpy precision" is that the range of z has to be restricted to include a single integer, whereas in fact the information that z is in the interval (10,15), say, might be all we need to know. A possible solution would therefore be to replace each term in a continued fraction by a sequence of intervals converging on the term. Thus instead of

```
[ 2,
  1000]
```

we might have

```
[ ((0,4), (1,3), 2),
  ((0,10000), (500,1400), (996, 1010), 1000) ]
```

This would give us 7 successive approximations to the result, instead of only 2. It is also fairly simple to adapt the method of the last section to use such a representation of continued fractions. Instead of assuming that x (and y) varies in (1,infinity), we use the interval given by the next element of x, and only ingest a term when we have consumed all the approximating intervals. Likewise, on the output, we can emit a better approximation to z whenever we compute its range, regardless of whether its integer part is constant. Finally, when z's integer part is closely enough bounded, we can egest a term; and then start computing approximations to the next term.

It should be noted that the endpoints of the intervals are integers (no fractional part is required). The whole system can then be regarded as an interval arithmetic package with variable "magnification", where each term of the continued fraction "turns up the magnification", so that we can still work with integers.

The net effect is to produce a new approximation to the final result of a calculation, however complex, in **constant time,** where the constant is determined only by the static complexity of the calculation, and not by the particular numbers involved. This must surely be as desirable a state of affairs as we could wish for.

One extremely pleasant consequence of using intervals in this way is that we can now represent approximate quantities by allowing the sequence of approximations to a term to stop before it has converged on a particular term. For instance 2.75 +- Ø.25 can be represented as [(2), ((1,2))], which says "my value is between [2,1] and [2,2]". The result will be computed to as much accuracy as can be guaranteed, and no more, giving an automatic error bound system for engineering calculations.

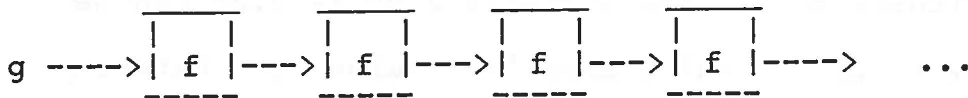
4.2 Transcendental functions

The author knows of no good way to compute functions such as exp(x), log(x), and sin(x). The ideal method would be based on the same principles as the one given for basic arithmetic.

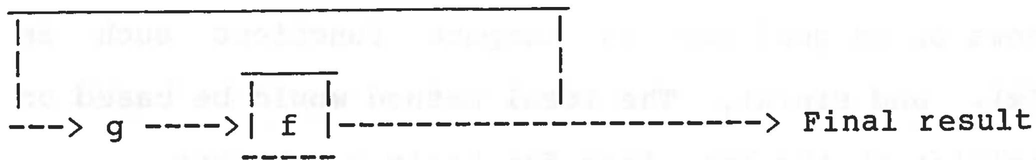
It is possible to use an infinite Taylor series, though at first sight it looks as if an infinite amount of work would need to be done to ensure that a late term did not swamp earlier ones (we can hardly expect the system to work out that the series is convergent!). This is easily dealt with, however, by appending on the front of the rest of the series an (analytically derived) interval giving a bound on the size of rest of the series - another benefit of using the interval method.

5. Iterative calculations

As a further illustration of the way a lazy functional language supports the use of arbitrary precision calculations, consider an iterative technique, such as Newton-Raphson, for finding a root of an equation. Typically we make a guess, g, of the root, and iteratively improve the guess by repeatedly applying some function, f, to successive guesses, halting the process when we believe we are sufficiently close to the root. We may illustrate this:



However, in our continued fraction world, we can feed back the terms of the improved guess (output from f) directly as the subsequent terms of the "previous guess" (input to f), thus:



The initial guess, g, must be an interval in this case. This feedback trick gives the terms of the final result directly, together with their implicit error bounds, without requiring the programmer to pay attention to how many iterations are required. It should be clear that lazy semantics are necessary for this trick to work.

6. Conclusions

As a novel application of a functional language providing lazy

evaluation, an arithmetic package has been described which has the following characteristics:

(i) Arbitrary precision real arithmetic.

(ii) Successive approximations to result produced in constant time.

(iii) Approximate quantities can be specified as inputs.

(iv) Results produced with guaranteed error bounds.

(v) No unnecessary work performed.

(vi) The ability to compute with infinite series.

A preliminary version of such a package has been implemented in Sasl [Turn], a functional programming language, and a more sophisticated version is under construction. While the cost of the operations is significant, the rewards are also, and the area merits further study, particularly to address the transcendental functions.

7. References

[Gosp80] "Continued fraction arithmetic". HAKMEM Item 101b.

[Gosp81] "Continued fraction arithmetic". Unpublished paper.

[Hard38] Hardy and Wright. "An introduction to the theory of numbers". Oxford 1960 (4th ed).

[Khin64] AY Khinchin. "Continued fractions". Chicago 1964.

[Turn] DA Turner. "The Sasl manual". University of Kent, England.

[Wied80] E Wiedmer. "Computing with infinite objects". Theoretical Computer Science 10 (1980), pp133-155, North Holland.

```
list sasd
>      1      def mul 'inf" b = 'inf"
>      2      mul a 'inf" = 'inf"
>      3      mul a b = a*b
>      4      add 'inf" b = 'inf"
>      5      add a 'inf" = 'inf"
>      6      add a b = a+b
>      7      sub 'inf" b = 'inf"
>      8      sub a 'inf" = 'inf"
>      9      sub a b = a-b
>     10      div 'inf" b = 'inf"
>     11      div a 0 = 'inf"
>     12      div a 'inf" = 0
>     13      div a b = a/b
>     14      remdr 'inf" b = 0
>     15      remdr a 0 = 0
>     16      remdr a b = a - (a/b)*b
>     17
>     18      cfrat 'inf" 'inf" = ()
>     19      cfrat 0 0 = ()
>     20      cfrat a b = (div a b) : (cfrat b (remdr a b))
>     20.1
>     21      ratcf x = x:(ratcf1 x)
>     21.1    ratcf1 h:t = (h,1):(ratcf2 t h 1 1 0)
>     21.2    ratcf2 'inf":() pm1 qm1 pm2 qm2 = ()
>     21.3    ratcf2 h:t pm1 qm1 pm2 qm2 = (p,q):(ratcf2 t p q pm1 qm1)
>     21.4                                  where p = h*pm1 + pm2
>     21.5                                        q = h*qm1 + qm2
>     21.61
>     22      expand coeffs x y = expandcv coeffs x y (cornervalues coeffs)
>     23      expandcv coeffs x y cvs = (alleq cvs) -> emit coeffs x y (hd cvs);
>     24                                              refinexy coeffs x y
>     25      refinexy coeffs xa:xb () = expand (refinex coeffs xa) xb ()
>     26      refinexy coeffs () ya:yb = expand (refiney coeffs ya) () yb
>     27      refinexy coeffs xa:xb ya:yb = expand (refinex (refiney coeffs ya) xa) xb yb
>     28
>     29      refinex (a,b,c,d,e,f,g,h) 'inf" = (0,0,a,b,0,0,e,f)
>     30      refinex (a,b,c,d,e,f,g,h) t =
>     31                                   (add (mul t a) c,
>     32                                    add (mul t b) d, a, b,
>     33                                    add (mul t e) g,
>     34                                    add (mul t f) h, e, f)
>     35
>     36      refiney (a,b,c,d,e,f,g,h) 'inf" = (0,a,0,c,0,e,0,g)
>     37      refiney (a,b,c,d,e,f,g,h) t =
>     37.1                                 (add (mul t a) b, a,
>     37.2                                  add (mul t c) d, c,
>     37.3                                  add (mul t e) f, e,
>     37.4                                  add (mul t g) h, g)
>     38
>     38.1    emit coeffs x y 'inf" = 'inf":()
>     39      emit coeffs x y t = t : (expand (shrug coeffs t) x y)
>     40      shrug (a,b,c,d,e,f,g,h) t =
>     41                                  (e, f, g, h,
>     42                                   sub a (mul t e),
>     43                                   sub b (mul t f),
>     44                                   sub c (mul t g),
>     45                                   sub d (mul t h))
>     46
>     47      cfadd x y = expand (0,1,1,0,0,0,0,1) x y
>     48      cfsub x y = expand (0,1,-1,0,0,0,0,1) x y
>     49      cfmul x y = expand (1,0,0,0,0,0,0,1) x y
>     50      cfdiv x y = expand (0,1,0,0,0,0,1,0) x y
>     51
>     51.1
>     51.2    cornervalues (0,0,0,d,0,0,0,h) = (w,w,w,w)
>     51.3                                     where w = div d h
>     51.4    cornervalues (0,0,c,d,0,0,g,h) = (v,w,v,w)
>     51.5                                     where v = div (add c d) (add g h)
>     51.6                                           w = div c g
>     51.61   cornervalues (0,b,0,d,0,f,0,h) = (v,v,w,w)
>     51.62                                    where v = div (add b d) (add f h)
>     51.63                                          w = div b f
>     52      cornervalues (a,b,c,d,e,f,g,h) =
>     53                                   (div (add a (add b (add c d)))
>     54                                        (add e (add f (add g h))),
>     55                                    div (add a c) (add e g),
>     56                                    div (add a b) (add e f),
>     57                                    div a e)
>     57.1
>     57.2    alleq (c11,c1x,cx1,cxx) = (c11=c1x) & (cx1=cxx) & (c11=cx1)
>     57.3
>     57.4    ?
>     58
>     59
>     60
>     61
#END OF FILE
#
```

```
%PLEASE ENTER TERMINAL TYPE.. tty43
%WHICH SERVICE? mts
%CALLING MTS
%CONNECTED TO MTS

MTS(01019)<=>N.CNTR2 (005)
#$sig yzf4
#Enter user password.
?XXXXX
#CHARGING RATE = UNIVERSITY, TERMINAL
#**LAST SIGNON WAS: 13:15:00
# USER "YZF4" SIGNED ON AT  14:09:42 ON FRI 31-JUL-81
>New 'phone nos. :- N'cle Comp Lab 329233, Poly 326002.
#$sou qcf0:sasl
#$CON *MSINK* LC
#$R YZB0:INTERSASL SCARDS=*MSOURCE* PAR='CORE=25P'QCF0:SASL.PRELUDE
#EXECUTION BEGINS
 New environment loaded from QCF0:SASL.PRELUDE
 The following funtions are defined:
 hd                tl                abs
 length            reverse           sum
 product           and               or
 count             from              map
 for               zip               while
 until             member            union
 intersection      digitval          spaces
 width             Ljustify          Rjustify
 Cjustify          show              append
 compose           digit             letter
 code              decode            list
 function          logical           char
 number


 hello from sasl
 8?
 8
 nu?
 get sasl

 nu?

 nu?

 nu?

 show (ratcf (cfadd (cfrat 45 34) (cfrat 253 17)))?
 ((16,4,1,6,'inf"),(16,1),(65,4),(81,5),(551,34))
 nu?
 show (ratcf (cfadd (cfrat 1427 1932) (cfrat (9253 3163)))?

 Syntax: ) expected where ? found in:

 show (ratcf (cfadd (cfrat 1427 1932) (cfrat (9253 3163)))?

 nu?
 show ?ratcf (cfadd (cfrat 1427 1932) (cfrat((9253 3162)))?

 Syntax: ) expected where ? found in:

 show (ratcf (cfadd (cfrat 1427 1932) (cfrat (9253 3162)))?

 nu?
 show (ratcf (cfadd (cfrat 295 396) (cfrat 826 534)))?
 ((2,3,2,2,1,16,1,3,2,2,1,2,'inf"),(2,1),(7,3),(16,7),(39,17),(55,24),(919,401),(974,425),(3841,1676),(8656,3777),(21153,9230),(298
 09,13007),(80771,35244))
 nu?
 show (ratcf (cfadd (cfrat 142 23) (cfrat 29 425)))?
 ((6,4,7,1,2,2,4,2,4,'inf"),(6,1),(25,4),(181,29),(206,33),(593,95),(1392,223),(6161,987),(13714,2197),(61017,9775))
 nu?
 show (ratcf (cfmul (cfrat 1234 3456) (cfrat  3241 3164)))?
 ((0,2,1,2,1,3,5,2,4,1,22,2,6,'inf"),(0,1),(1,2),(1,3),(3,8),(4,11),(15,41),(79,216),(173,473),(771,2108),(944,2581),(21539,58890),
 (44022,120361),(285671,781056))
 nu?
 show (ratcf (cfdiv (cfrat 147 297) (cfrat 425 924)))?
 ((1,13,6,1,13,'inf"),(1,1),(14,13),(85,79),(99,92),(1372,1275))
 nu?
 show(cfadd (cfrat 123 456) (cfrat 789 123)))?
 (6,1,2,5,1,16,2,2,1,2,'inf")
 nu?

 Syntax: Expression expected where ? fo[illegible] in:



 nu?
 show (hd (cfadd (cfrat 123 456) (cfrat([illegible]9 123)))?
 6
 nu?
```