Implementing Infinite Precision Arithmetic

Jerry Schwarz
AT&T Bell Labs
Murray Hill, NJ

Abstract

A data structure for exact representation of real numbers is presented. The representation allows exact computation involving ordinary arithmetic operations on rationals, irrationals and even some transcendental values (such as π). Functions defined via infinite series may also be exactly evaluated. Algorithms are described and analyzed. An implementation in C++ is described.

1 Introduction

There are two common ways for higher level languages to support numeric computations when hardware representations of numbers have inadequate precision. The "multiple precision" approach (for example in Numerical Turing[1]) uses a floating point representation with more bits and implements the operations in software. The "bignum" approach (for example in LISP[2]) uses dynamically allocated storage to represent arbitrarily large integers. A drawback of the multiple precision approach is that numbers still have finite precision. The problem with bignum representations is that, even when extended to deal with rational combinations of integers, they cannot represent irrational values, such as $\sqrt{2}$. This paper presents a representation that combines the properties of floating point and arbitrary precision. A number is represented partially as a sequence of bits and partially as an "expression" that represents bits that have not yet been "expanded". When a demand for more bits is generated, the expression will be manipulated until the demanded bits are determined. Typically, demand is generated by the need to compare two numbers or to output a number.

Exact representations of numbers have been described by Boehm et. al.[3], Boehm[4] and Vuillemin[5]. Of these, the methods described in this paper are closest to that of Boehm[3]. They differ primarily in the way that a demand for bits propagates from an expression to subexpressions. In Boehm[3] the demand propagated in fixed patterns. In the methods of this paper the values of subexpressions are considered in propagating demand.

The representation described in this paper has been used in a C++ library as described in Schwarz[6]. The representation is intended for use in situations where the cost of exact computation is worth the expense. For example

- When there is some part of a computation in which high precision is required. In this case, the program might use exact numbers in some places and floating point types in others.
- When an analyst wants to determine if a program using ordinary floating point is suffering from a numerical instability.
- When an analyst wants to compute some numbers to high precision, either because of intrinsic interest or to serve as reference values for methods using ordinary floating point numbers.

Some basic concepts used in this paper:

real number: The ordinary mathematical notion of real number. Real numbers are written in italics thus: x, y, k,....

small value: A real number x is small if it satisfies $-\mathcal{B} \leq x \leq \mathcal{B}$, where $\mathcal{B}$ is an arbitrary "base" that will play a major role in the representations discussed in this paper.

bigit(rhymes with digit): An integer value b satisfying $-\mathcal{B} < b < \mathcal{B}$. The name comes from "big digit". Notice that bigits may be negative. Bigits are written in italics thus: a, b,....

primitive expression: A data structure used to represent small values. Primitive expressions are written in bold font thus **x**, **y**,.... Primitive expressions are built from bigits and (pointers to) other primitive expressions, just as an arbitrary expression would be. However only a limited repertoire of combining forms is allowed.

exact number: A data structure used to represent arbitrary real values. It consists of an integer exponent and a primitive expression.

Consider a real value $\mathbf{x}$ represented in a maximally redundant balanced base $\mathcal{B}$ notation.

$$x = \mathcal{B}^e \sum_{i=0}^{\infty} b_i \mathcal{B}^{-i}$$

where the exponent e is an integer, and $\langle b_i \rangle$ is a sequence of bigits. A real value can be broken down in this way into many combinations of exponents and smalls. And a small, in turn, has many different representations as a sequence of bigits. Redundancy is inherent in the problem. Any representation in which exact computations are possible with certain other desirable properties (see Vuillemin[5]) must exhibit redundancy.

Management of exponents is straightforward, so the central concern of this paper is to finds ways of representing primitive expressions that permit them to be efficiently manipulated and that control the amount of work required to expand them into sequences of bigits.

2 Primitive Expressions

It is necessary to clearly distinguish between the primitive expressions and the real values they represent. I use an applicative style to notate primitive expressions and I write a bar over the data structure to indicate the value it represents.

The simplest primitive expression is $\mathtt{expanded}(b, \mathbf{x})$, which contains a bigit and a subexpression.

$$\overline{\mathtt{expanded}(b, x)} = b + \overline{x}\mathcal{B}^{-1}$$

expanded is so important that there is a special infix notation for it.

$$\mathtt{expanded}(b, \mathbf{x}) \quad = \quad b \oplus \mathbf{x}$$

Taking an arbitrary primitive expression and finding an equivalent (representing the same value) **expanded** expression is called *expansion.* b is called the integer part and $\mathbf{x}$ the fractional part of $b \oplus \mathbf{x}$. $\oplus$ associates to the right so, $a \oplus b \oplus \mathbf{x}$ is the same as $a \oplus (b \oplus \mathbf{x})$.

The following representations are used, where b and the multiplier m are bigits, $\mathbf{x}$ and $\mathbf{y}$ are subexpressions representing smalls, a carry c is an integer satisfying $|c| < \mathcal{B}^2$, and a shift s is a non-negative integer.

$$\begin{array}{lcl}
\overline{\mathtt{abs}(\mathbf{x})} & = & |\overline{\mathbf{x}}| \\
\overline{\mathtt{cprod}(c, \mathbf{x}, \mathbf{y})} & = & c + \overline{\mathbf{x}}\,\overline{\mathbf{y}} \\
\overline{\mathtt{expanded}(b, \mathbf{x})} & = & b + \overline{\mathbf{x}}\,\mathcal{B}^{-1} \\
\overline{\mathtt{minus}(\mathbf{x})} & = & -\overline{\mathbf{x}} \\
\overline{\mathtt{mult}(c, m, \mathbf{x})} & = & c + m\,\overline{\mathbf{x}} \\
\overline{\mathtt{prod}(\mathbf{x}, \mathbf{y})} & = & \overline{\mathbf{x}}\,\overline{\mathbf{y}}\,\mathcal{B}^{-1} \\
\overline{\mathtt{sum}(c, \mathbf{x}, \mathbf{y})} & = & c + \overline{\mathbf{x}} + \overline{\mathbf{y}} \\
\overline{\mathtt{quot}(c, \mathbf{x}, \mathbf{y})} & = & (c + \overline{\mathbf{x}})/\overline{\mathbf{y}} \\
\overline{\mathtt{shift}(\mathbf{x}, s)} & = & \mathcal{B}^{-s}\,\overline{\mathbf{x}} \\
\overline{\mathtt{zero}} & = & 0
\end{array}$$

There is no theoretical basis for this choice of primitive expressions. Many other collections would have served. They were chosen because they provided a convenient collection that "fit together" in a way that allows expansion to work. (See section 4.)

In some cases the value represented by a primitive expression will be a small value for any (small) values of its operands. This is true for **expanded**, **minus**, **zero**, **prod** and **abs**. In the other cases for certain operands the value would be too positive or too negative. In what follows primitive expressions will only be written when it can be shown that the represented value is in the proper range. For example, in order to replace $\overline{x}\,\overline{y}$ by $\overline{\mathtt{cprod}(0, \mathbf{x}, \mathbf{y})}$ it would be necessary to show that $-\mathcal{B} \le \overline{x}\,\overline{y} \le \mathcal{B}$.

To help the reader understand how primitive expressions are manipulated it is useful to consider some examples. To make the numbers easier to follow the the rest of this section assumes that $\mathcal{B} = 100$.

Consider

$$\mathtt{sum}(0, \mathtt{prod}(55, 11 \oplus \mathbf{x}), \mathtt{mult}(-500, 18, 29 \oplus \mathbf{y}))$$

Substituting from the above equations shows that this represents the value

$$\begin{array}{l}
(55(11 + \overline{\mathbf{x}}/100)/100) + (-500 + 18(29 + \overline{\mathbf{y}}/100)) \\
= 28.05 + .0055\overline{\mathbf{x}} + .18\overline{\mathbf{y}}
\end{array}$$

Because $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ are known to be small values (i.e. have absolute value at most 100) this value must be between -12.5 and 46.6 Suppose we want to determine the integer part. We first determine bounds on $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$. Suppose we examine $\mathbf{x}$ and find that $\overline{\mathbf{x}}$ is bounded by -15 and -5. This means $28.05 + .0055\overline{\mathbf{x}}$ is bounded by 27.9675 and 28.0225. Dropping extra bigits past the decimal point the bounds can be stated as 27.96 and 28.03. That is, the possible values of $28.05 + .0055\overline{\mathbf{x}}$ have a range less than .07. If we can determine the value of $.15\overline{\mathbf{y}}$ to within less than .93 we would have an uncertainty of less than 1.0 in the overall number and this will allow us to determine an integer part for the overall number. If examination of $\mathbf{y}$ does not immediately yield a range of 6 we will manipulate it until it has such a range. This process is called

narrowing. If all else fails we may have to expand **y** into its integer and fractional parts. However, frequently we can manipulate the primitive expression so as to determine bounds within the desired range without expanding it. Assume that such manipulation yields bounds on $\overline{\mathbf{y}}$ of 20 and 25. This yields bounds on the value of 31.56 and 32.53. implying an integer part of 32. To determine a primitive expression for the fractional part we work backwards. To reach 32 we used 6 from $55(11+\overline{\mathbf{x}}/100)$ and 26 from $-500+15(29+\overline{\mathbf{y}}/100)$. We can rearrange the first of these as $6+\mathtt{mult}(-600, 55, 11\oplus\mathbf{x})$ and the second as $26+\mathtt{mult}(-400, 18, \mathbf{y})$ to give the final expansion as

$$32 \oplus \mathtt{sum}(0, \mathtt{mult}(-600, 55, 11 \oplus \mathbf{x}), \mathtt{mult}(-400, 18, \mathbf{y}))$$

3 Algorithmic Notations

The algorithms in this paper are presented in an equational style. The left hand side of each equation is a "pattern" specifying a condition, and another condition involving values may optionally be specified with an explicit "if". The intention is that these can be translated into an ordinary recursive function in a language such as LISP or ML.

For example, consider a (nonsense) function **ugly** which has equations:

$$\begin{array}{lcl} \mathtt{ugly}(\mathtt{sum}(a, \mathtt{zero}, \mathbf{y})) & = & 1+\mathtt{ugly}(\mathbf{y}) \text{ if } a=0 \\ \mathtt{ugly}(\mathtt{sum}(a, \mathbf{x}, \mathtt{zero})) & = & 1+\mathtt{ugly}(\mathbf{x}) \text{ if } a=0 \\ \mathtt{ugly}(\mathtt{prod}(\mathbf{x}, \mathbf{y})) & = & 0 \\ \mathtt{ugly}(a \oplus \mathbf{x}) & = & a+\mathtt{ugly}(\mathbf{x}) \end{array}$$

Within a block, the equations are regarded as ordered so the conditions are not neccessarily exclusive. When equations for a funtion are presented in multiple blocks (usually scattered in several positions in this paper) they will (except as noted) treat independent cases and the order of blocks will not matter. An equation containing $\oplus$ on the left hand side has a special meaning. It represents a method to be used if no other cases match. This consists of calling **expand** on the argument before attempting to use it. Since (see section 6) a call of **expand** always terminates with an **expanded**, such a pattern will always succeed. Hence it should always be ordered as the last case

4 Expansion

Computation on exact numbers consists primarily of building primitive expressions and expanding them. The later is done by a function **expand** that takes a primitive expression and yields an **expanded** primitive expression satisfying

$$\overline{\mathtt{expand}(\mathbf{x})} = \overline{\mathbf{x}}$$

The full expansion of a primitive expression is determined by expanding it, expanding the fractional part, expanding the fractional part of that, and so on. Although **expand** always terminates, the process of examining the fractional parts may proceed indefinitely. Thus the representation can deal with arbitrary (computable) real values.

Two important special functions for picking apart the integer and fractional parts of primitive expressions get their own special notation

$$\begin{array}{lcl} [a \oplus \mathbf{x}]_{ipart} & = & a \\ [a \oplus \mathbf{x}]_{fpart} & = & \mathbf{x} \end{array}$$

Notice that by the conventions of section 3 these equations imply that a call of **expand** will be performed when neccessary.

The definition of **expand** uses some auxiliary functions satisfying

$$\begin{array}{l} -\mathcal{B} < \mathtt{leading}(\mathbf{x}, \mathbf{y}) < \mathcal{B} \\ -\mathcal{B} < \mathtt{carry}(\mathbf{x}, \mathbf{y}) < \mathcal{B} \\ -\mathcal{B} < \mathtt{carry}(\mathbf{x}, \mathbf{y}, \mathbf{z}) < \mathcal{B} \\ \overline{\mathbf{x}\,\mathbf{y}} = \mathtt{leading}(\mathbf{x}, \mathbf{y})\,\mathcal{B} + \overline{\mathtt{low}(\mathbf{x}, \mathbf{y})} \\ \overline{\mathbf{x}+\mathbf{y}} = \mathtt{carry}(\mathbf{x}, \mathbf{y})\,\mathcal{B} + \overline{\mathtt{plus}(\mathbf{x}, \mathbf{y})} \\ \overline{\mathbf{x}+\mathbf{y}+\mathbf{z}} = \mathtt{carry}(\mathbf{x}, \mathbf{y}, \mathbf{z})\,\mathcal{B} + \overline{\mathtt{plus}(\mathbf{x}, \mathbf{y}, \mathbf{z})} \\ \text{if } \overline{\mathbf{y}} \geq \mathcal{B}/2 \text{ then } \left|\mathtt{approxq}(c, \mathbf{x}, \mathbf{y}) - \dfrac{c+\overline{\mathbf{x}}}{\overline{\mathbf{y}}}\right| < 1 \\ \text{if } \overline{\mathbf{y}} \geq \mathcal{B}/2 \text{ then } \dfrac{c+\overline{\mathbf{x}}}{\overline{\mathbf{y}}} = \dfrac{\mathtt{approxq}(c, \mathbf{x}, \mathbf{y})}{+\mathtt{trailing}(c, \mathbf{x}, \mathbf{y})\,\mathcal{B}^{-1}} \end{array}$$

leading, **carry** (two and three argument versions) and **approxq** return bigits. **low**, **plus** (two and three argument versions) and **trailing** return primitive expressions. Functions taking primitive expressions can be extended to accept bigits by using the primitive expression $a \oplus \mathtt{zero}$ in place of a.

The equations for **expand** are displayed in figure 1. Immediate inspection shows that the right hand side of each of these equations is an **expanded** expression. To show that the right hand side satisfies the requirements on **expand** the following must also be shown. (Termination is discussed in section 6)

- The right hand side represents the same value as the left hand side. These are simple algebraic exercises using the assumed properties of the auxiliary functions.
- The fractional parts represent small values. Again simple algebraic manipulations suffice, using the

$$
\begin{aligned}
\texttt{expand}(\texttt{abs}(b \oplus \mathbf{x})) &= 0 \oplus \texttt{abs}(\mathbf{x}) \text{ if } b = 0 \\
\texttt{expand}(\texttt{abs}(b \oplus \mathbf{x})) &= b \oplus \mathbf{x} \text{ if } b > 0 \\
\texttt{expand}(\texttt{abs}(b \oplus \mathbf{x})) &= -b \oplus \texttt{minus}(\mathbf{x}) \text{ if } b < 0 \\
\texttt{expand}(\texttt{cprod}(c, a \oplus \mathbf{x}, b \oplus \mathbf{y})) &= (c + ab + \texttt{leading}(b, \mathbf{x}) + \texttt{leading}(a, \mathbf{y}) \\
&\quad + \texttt{carry}(\ \texttt{mult}(-\texttt{leading}(b, \mathbf{x})\mathcal{B}, b, \mathbf{x}), \\
&\qquad \texttt{mult}(-\texttt{leading}(a, \mathbf{y})\mathcal{B}, a, \mathbf{y}), \\
&\qquad \texttt{prod}(\mathbf{x}, \mathbf{y})) \\
&\quad \oplus \texttt{plus}(\ \texttt{mult}(-\texttt{leading}(b, \mathbf{x})\mathcal{B}, b, \mathbf{x}), \\
&\qquad \texttt{mult}(-\texttt{leading}(a, \mathbf{y})\mathcal{B}, a, \mathbf{y}), \\
&\qquad \texttt{prod}(\mathbf{x}, \mathbf{y})) \\
\texttt{expand}(\texttt{minus}(b \oplus \mathbf{x})) &= -b \oplus \texttt{minus}(\mathbf{x}) \\
\texttt{expand}(\texttt{mult}(c, m, b \oplus \mathbf{x})) &= (c + m\,b + \texttt{leading}(m, \mathbf{x})) \\
&\quad \oplus \texttt{mult}(-\texttt{leading}(m, \mathbf{x})\mathcal{B}, m, \texttt{low}(m, \mathbf{x})) \\
\texttt{expand}(\texttt{prod}(\mathbf{x}, \mathbf{y})) &= \texttt{leading}(\mathbf{x}, \mathbf{y}) \oplus \texttt{low}(\mathbf{x}, \mathbf{y}) \\
\texttt{expand}(\texttt{quot}(c, \mathbf{x}, \mathbf{y})) &= \texttt{approxq}(\mathbf{x}) \oplus \texttt{trailing}(c, \mathbf{x}, \mathbf{y}) \\
\texttt{expand}(\texttt{sum}(c, a \oplus \mathbf{x}, b \oplus \mathbf{y})) &= (c + a + b + \texttt{carry}(\mathbf{x}, \mathbf{y})) \oplus \texttt{plus}(\mathbf{x}, \mathbf{y}) \\
\texttt{expand}(\texttt{zero}) &= 0 \oplus \texttt{zero}
\end{aligned}
$$

Figure 1: definition of **expand**

properties of the auxiliary functions and also the assumption that the subexpressions of the argument to **expand** represent smalls.

- The integer parts i satisfy $-\mathcal{B} < i < \mathcal{B}$. This is a bit tricky. (And requires some modification to the above equations.) Since the fractional parts, $\mathbf{z}$ are known (by the previous item) to represent small values, $\mathcal{B}^{-1}\,\overline{\mathbf{z}}$ is between -1 and and 1. Since the right hand side is known to represent the same value as the argument of **expand**, which is assumed to represent a small value, this implies $-\mathcal{B} - 1 \leq i \leq \mathcal{B} + 1$. If $i = \mathcal{B} + 1$ then the only possible value that the argument of expand could represent is exactly $\mathcal{B}$. The author has been unable to construct an example in which $i = \mathcal{B} + 1$ would arise, but he has also been unable to prove it cannot. If it does then the expanded value could be replaced by an appropriate primitive expression. If $i = B$ then $\mathbf{z}$ must represent a negative value and we may replace the expanded value by $(\mathcal{B} - 1) \oplus \texttt{sum}(\mathcal{B}, \mathbf{z}, \texttt{zero})$.

  The above equations must be replaced by equations that incorporate some additional transformations in the above special cases.

  Similar manipulations are necessary when the integer part would be $-\mathcal{B} - 1$ or $-\mathcal{B}$.

4.1 Details of Addition

This section discusses the details of expanding addition. Addition is the simplest (binary) operation. Multiplication (in its various forms) and division follow essentially the same pattern, although the details are more complicated.

The fundamental function is **carry**, the function that determines the first bigit in the expansion of a sum. The function **plus** is defined in terms of it.

$$
\begin{aligned}
\texttt{plus}(\mathbf{x}, \mathbf{y}) &= \texttt{sum}(-\texttt{carry}(\mathbf{x}, \mathbf{y})\mathcal{B}, \mathbf{x}, \mathbf{y}) \\
\texttt{plus}(\mathbf{x}, \mathbf{y}, \mathbf{z}) &= \texttt{plus}(\mathbf{x}, \texttt{plus}(\mathbf{y}, \mathbf{z}))
\end{aligned}
$$

carry is itself defined in terms of other functions. Two of these are **lbound** and **ubound** which satisfy.

$$-\mathcal{B} \leq \texttt{lbound}(\mathbf{x}) \leq \overline{\mathbf{x}} \leq \texttt{ubound}(\mathbf{x}) \leq \mathcal{B}$$

It is required that **lbound** and **ubound** always terminate, and in practice it is important that they terminate quickly without expanding any subexpressions. **lbound** and **ubound** might return arbitrary reals between $-\mathcal{B}$ and $\mathcal{B}$, but we need to represent and manipulate their values directly. In practice a fixed point representation is convenient. The real numbers returned by **lbound** and **ubound** (and manipulated by the function **narrow** discussed below) are represented by integers. Except for a tricky place in division where a couple of extra bits are required, a single bigit of precision after the "bigit point" suffices. There are a lot of straightforward cases to be defined in ways that resemble interval arithmetic. The complete definition is omitted, but some typical cases are:

$$
\begin{aligned}
&\texttt{lbound}(\texttt{abs}(\mathbf{x})) \\
&\quad = \max(0, \texttt{lbound}(\mathbf{x}), -\texttt{ubound}(\mathbf{x})) \\
&\texttt{ubound}(\texttt{sum}(c, x, y)) \\
&\quad = \min(\mathcal{B}, c + \texttt{ubound}(\mathbf{x}) + \texttt{ubound}(\mathbf{y}))
\end{aligned}
$$

range is defined in terms of **lbound** and **ubound**

$$\mathbf{range(x)} = \mathbf{ubound(x)} - \mathbf{lbound(x)}$$

The final function required to define **carry** is **narrow** which in many ways distinguishes the approach in this paper from others. Assuming $n \neq 0$ it satisfies:

$$\overline{\mathbf{narrow(x,}n\mathbf{)}} = \overline{\mathbf{x}}$$
$$\mathbf{range(narrow(x,}n\mathbf{))} \leq n$$

narrow may return a primitive expression similar to **x** with narrowed subexpressions or in "tough" cases expand it. In practice, it is important that **narrow** not only return a new primitive expression, but that it also modify the data structure that is its argument. This avoids repeating work later. **narrow** is described further in section 4.2

The cases in which **carry(x, y)** can be immediately determined are

$$\begin{aligned}
\mathbf{carry(x,y)} &= 0 \text{ if } \mathbf{ubound(x)} + \mathbf{ubound(y)} < \mathcal{B} \\
&\quad \text{and} \\
&\quad \mathbf{lbound(x)} + \mathbf{lbound(y)} > -\mathcal{B} \\
\mathbf{carry(x,y)} &= 1 \text{ if } \mathbf{ubound(x)} + \mathbf{ubound(y)} > 0 \\
\mathbf{carry(x,y)} &= -1 \text{ if } \mathbf{lbound(x)} + \mathbf{lbound(y)} < 0
\end{aligned}$$

It pays to give the value 0 preference because doing so avoids creating extra bigits in expansions that are later removed. (This is discussed further is section 5)

In case none of the above apply to **carry(x, y)**, narrowing of one or both of the subexpressions is required before one of the above equations can apply.

For more details of addition and details of multiplication and division the reader is referred to a long form of the paper available from the author.

4.2 Narrowing

narrow(x, n**)** is defined via a case analysis which either determines the value or narrows some subexpression and tries again. n must, of course, be non-zero.

The immediate cases are:

$$\begin{aligned}
\mathbf{narrow(x,}n\mathbf{)} &= \mathbf{x} \text{ if } n \geq 2\,\mathcal{B} \\
\mathbf{narrow(x,}n\mathbf{)} &= \mathbf{x} \text{ if } \mathbf{range(x)} \leq n \\
\mathbf{narrow(x,}n\mathbf{)} &= [\mathbf{x}]_{ipart} \oplus \mathbf{narrow}([\mathbf{x}]_{fpart}, n\,\mathcal{B}) \\
&\quad \text{if } n \leq 2
\end{aligned}$$

If non of the above equations is applicable, other cases must be dealt with by narrowing some subexpression of **x** and calling **narrow** recursively.

$$\mathbf{narrow(sum(}c\mathbf{,x,y)},n\mathbf{)}$$

is dealt with according to the following scheme for narrowing:

$$\begin{array}{lcll}
\mathbf{y} & \triangleright & n - \mathbf{range(x)} - 1 & \text{if } 2 < \mathbf{range(x)} < (n/2) \\
\mathbf{x} & \triangleright & n - \mathbf{range(y)} - 1 & \text{if } 2 < \mathbf{range(y)} < (n/2) \\
\mathbf{x} & \triangleright & n/2 & \text{if } \mathbf{range(x)} \geq (n/2) \\
\mathbf{y} & \triangleright & n/2) & \text{if } \mathbf{range(y)} \geq (n/2)
\end{array}$$

where $\mathbf{x} \triangleright m$ is an abbreviation for a right hand side **narrow(sum(**c**, narrow(x,** m**), y)** and similarly for $\mathbf{y} \triangleright m$. Cases for other primitive expressions are omitted due to space considerations.

4.3 Summary of Expansion

The process of expanding a primitive expression **x** can be summarized

- See if the required bigit can be determined just by examining "immediate" bounds on the values of the subexpressions.
- If not, narrow a subexpression of **x** until the bigit can be determined. (This is the role of **carry** and **leading**)
- Return the **expanded** expression consisting of the determined bigit, and a fraction built from the subexpressions and the determined bigit. No further expansion should be carried out. The various sorts of primitive expressions must "fit together" properly in order for this to be possible in all circumstances.

5 Representing All Reals

A primitive expression must be combined with an exponent to represent an arbitrary real number. Such a data structure, called an *exact number*, can represent an arbitrary real number.

Manipulations are possible using the definition shown in figure 2. This figure uses the notation $\mathbf{x} \gg s$ for **shift(x,** s**)**.) If the last case in the definition of $\div$ is applicable the second case will apply to the recursive call, but no such guarantee applies to the first case. In this representation an attempt to construct the (non-primitive) expression that results from a division by zero does not terminate.

The special cases for **shift** are essential in dealing with multiplication. Without them, much effort is expended expanding leading 0 bigits.

It is important to deal with absolute value directly (with its own primitive expression) rather than attempting to do a comparison to zero. (See section 8.)

$$
\begin{array}{lcll}
\mathcal{B}^e\mathbf{x} + \mathcal{B}^f\mathbf{y} & = & \mathcal{B}^e \oplus \mathbf{sum}(0,\mathbf{x},\mathbf{y}) & \text{if } e = f \\
 & & & \quad \text{and } \mathbf{carry}(\mathbf{x},\mathbf{y}) = 0 \\
\mathcal{B}^e\mathbf{x} + \mathcal{B}^f\mathbf{y} & = & \mathcal{B}^{e+1}(\ \mathbf{carry}(\mathbf{x},\mathbf{y}) & \text{if } e = f \\
 & & \quad \oplus \mathbf{sum}(-\mathbf{carry}(\mathbf{x},\mathbf{y})\mathcal{B},\mathbf{x},\mathbf{y})) & \\
\mathcal{B}^e\mathbf{x} + \mathcal{B}^f\mathbf{y} & = & \mathcal{B}^e\mathbf{x} + \mathcal{B}^e(\mathbf{y} \gg (e-f)) & \text{if } e > f \\
\mathcal{B}^e\mathbf{x} + \mathcal{B}^f\mathbf{y} & = & \mathcal{B}^f(\mathbf{x} \gg (f-e)) + \mathcal{B}^f\mathbf{y} & \text{if } e < f \\
\mathcal{B}^e(\mathbf{x} \gg s) + \mathcal{B}^f(\mathbf{y} \gg t) & = & \mathcal{B}^{e-s}\mathbf{x} + \mathcal{B}^{f-t}\mathbf{y} & \text{if } e-s = f-t \\
\mathcal{B}^e(\mathbf{x} \gg s) + \mathcal{B}^f(\mathbf{y} \gg t) & = & \mathcal{B}^{e-s}\mathbf{x} + \mathcal{B}^{e-s}(\mathbf{y} \gg ((e-s)-(f-t))) & \text{if } e-s > f-t \\
\mathcal{B}^e(\mathbf{x} \gg s) + \mathcal{B}^f(\mathbf{y} \gg t) & = & \mathcal{B}^{f-t}(\mathbf{x} \gg ((f-t)-(e-s))) + \mathcal{B}^{f-t}\mathbf{y} & \text{if } e-s < f-t \\
 & & & \\
\mathcal{B}^e\mathbf{x} \times \mathcal{B}^f\mathbf{y} & = & \mathcal{B}^{e+f+1}\mathbf{prod}(\mathbf{x},\mathbf{y}) & \\
 & & & \\
\mathcal{B}^e\mathbf{x} \div \mathcal{B}^f(b \oplus \mathbf{y}) & = & -(\mathcal{B}^e\mathbf{x} \div \mathcal{B}^f(-b \oplus \mathbf{minus}(\mathbf{y}))) & \text{if } b < 0 \\
\mathcal{B}^e\mathbf{x} \div \mathcal{B}^f(b \oplus \mathbf{y}) & = & \mathcal{B}^e\mathbf{x} \div \mathcal{B}^{f-1}\mathbf{y} & \text{if } b = 0 \\
\mathcal{B}^e\mathbf{x} \div \mathcal{B}^f(b \oplus \mathbf{y}) & = & (\mathcal{B}^e\mathbf{x} \times \left\lfloor \frac{\mathcal{B}-1}{b} \right\rfloor) \div (\mathcal{B}^f\mathbf{y} \times \left\lfloor \frac{\mathcal{B}-1}{b} \right\rfloor) & \text{if } \mathcal{B}/2 > b > 0 \\
\mathcal{B}^e\mathbf{x} \div \mathcal{B}^f(b \oplus \mathbf{y}) & = & \mathcal{B}^{e-f}\mathbf{quot}(0,\mathbf{x},\mathbf{y}) & \text{if } b \geq \mathcal{B}/2 \\
 & & & \\
-\mathcal{B}^e\mathbf{x} & = & \mathcal{B}^e\mathbf{minus}(\mathbf{x}) & \\
|-\mathcal{B}^e\mathbf{x}| & = & \mathcal{B}^e\mathbf{abs}(\mathbf{x}) &
\end{array}
$$

Figure 2: definition of operations for exact numbers

6 Termination of Expansion

Call a primitive expression $\mathbf{x}$ 1-expandable if $\mathbf{expand}(\mathbf{x})$ terminates. Call it k-expandable (for $k > 1$) if $\mathbf{expand}(\mathbf{x})$ terminates and $[\mathbf{x}]_{fpart}$ is $(k-1)$-expandable. That is, a primitive expression is k-expandable if it is possible to compute the first k bigits of its fully expanded form.

The essense of the proof is to show that in order to expand a primitive expression it is neccessary to look only at no more than two bigits expansion of any subexpression. More details are contained in the long version of this paper

7 Infinite Series

The representation and algorithms presented so far may have seemed to the reader as an elaborate and inefficient way to do arithmetic on rational values. Nothing said so far has indicated how to create primitive expressions for irrational values. This section describes how the representation is extended to irrationals and even transcendental values (such as $\sqrt{2}$ or π).

Suppose we want to represent an infinite sum,

$$x = \sum_{n=0}^{\infty} \frac{1}{n!}$$

(which happens to be e, the base of natural logarithms.) This is close enough to an expansion in terms of bigits that we can imagine creating a new primitive expression for it. Something like

$$
\begin{array}{lcl}
\overline{\exp(k)} & = & \sum_{n=k}^{\infty}(1/n!) \\
\mathbf{lbound}(\exp(k)) & = & 0 \\
\mathbf{ubound}(\exp(k)) & = & 3/k! \\
\mathbf{expand}(\exp(k)) & = & \ldots
\end{array}
$$

Even before we try to address expansion we notice a problem. **ubound** gets very small and since we represent bounds with a limited precision it is quickly going to become worthless. Also we notice that even if we could make this work, the expansion of $\exp(k)$ for even moderate size k's would have a lot of leading zero bigits. There would certainly have to be a way to keep track of these. So more reasonable definitions are

$$
\begin{array}{lcl}
\overline{\mathbf{exp}(k,s)} & = & \mathcal{B}^s \sum_{n=k}^{\infty}(1/n!) \\
\mathbf{lbound}(\mathbf{exp}(k,s)) & = & 0 \\
\mathbf{ubound}(\mathbf{exp}(k,s)) & = & 3\mathcal{B}^s/k! \\
\mathbf{expand}(\mathbf{exp}(k,s)) & = & 0 \oplus \mathbf{exp}(k,s+1) \\
 & & \quad \text{if } \mathbf{ubound}(\mathbf{exp}(k,s)) < 1 \\
\mathbf{expand}(\mathbf{exp}(\mathbf{k},s)) & = & \mathbf{sum}(0, 1 \div k!, \mathbf{exp}(k+1,s)) \\
 & & \quad \text{if } \mathbf{ubound}(\mathbf{exp}(k,s)) < \mathcal{B}
\end{array}
$$

The **ubound** expression treats k as a real number, while the last equation for **expand** treats $1 \div k!$ as a primitive expression. An implementation would use an exact

number and look into the representation as necessary. None of the manipulations required are difficult. The comparison in the first case of **expand** can be done with a generous tolerance (see section 8) because if it fails the other case can be used without any problems. As usual, we are careful to only construct **exp** expressions which represent a small. And we are careful that the arguments of the **sum** represent smalls.

There is still a severe problem with the **exp** primitive expression. Recomputing $k!$ over and over will be expensive. Avoiding that recomputation is a simple exercise that is left to the reader.

Having seen how to deal with one infinite series it is easy to generalize to any other infinite series for which a bound on the sum of remaining terms can be be computed. For example an exact number representing π is easy to create. Using exact numbers to support the implementation of primitive expressions is a useful trick for handling bookkeeping.

8 Rounding and Comparisons

An important operation that has not yet been discussed is testing exact numbers for equality to zero, or more generally comparing them to zero. In the presense of representations for series as in section 7 it is not always possible to determine if a primitive expression is zero.

The basic function is **sgn** which satisfies

$$\begin{array}{ll} \text{if } \mathbf{sgn}(\mathbf{x}) = 0 & \text{then } \overline{\mathbf{x}} = 0 \\ \text{if } \mathbf{sgn}(\mathbf{x}) < 0 & \text{then } \overline{\mathbf{x}} < 0 \\ \text{if } \mathbf{sgn}(\mathbf{x}) > 0 & \text{then } \overline{\mathbf{x}} < 0 \end{array}$$

These are one way implications, $\mathbf{sgn}(\mathbf{x})$ may not be defined. In particular when $\mathbf{x}$ is a primitive expression that would expand to an infinite sequence of 0 bigits, then it may not be possible to know this and **sgn** could run on forever. So **sgn** is given tolerance k whose purpose is to indicate how much of $\mathbf{x}$ to examine before giving up. If $|\overline{\mathbf{x}}| > \mathcal{B}^{-k}$ then $\mathbf{sgn}(\mathbf{x}, k)$ gives the correct result.

What should happen in the case that "reports failure" depends on how **sgn** is being used. In some instances it does represent a failure and computation should be stopped. In other cases an arbitrary value (0, 1 or -1) may be chosen.

Rounding or truncating are operations closely related to comparison, but here the tolerance is used to indicate the proper result when **sgn** would report failure.

9 Implementation Issues

The algorithms described in this paper have been implemented as a C++[7] library. The interface to that library is described in Schwarz[6]. It consists of a defined class(type) which to the programmer is very similar to ordinary floating point types. The main difficulties in the interface have to do with the tolerance issues discussed in section 8. It should be emphasized that the library is written entirely in C++ and the interface is an ordinary C++ class declaration. No changes were necessary to the language. This section describes some of the details of the implementation.

The major distinction between the implementation and the algorithms described in this paper is that the equations presented here are somewhat cavalier about repeated expressions, while the implementation must not be so cavalier. This applies to the individual cases (where computation is frequently repeated in conditions) and to the overall structure. In the implementation each primitive expression is represented by an object(i.e. a record or struct) of a C++ class, Small. Subexpressions are represented as pointers to other Smalls. Expansion and narrowing are done by modifying the previously allocated Small so that any other Smalls that point to it see the improved representation. Bounds(`lbound` and **ubound**) are computed and stored in the Small when it is allocated and when it is expanded or narrowed. If a narrowing or expansion of a subexpression occurs it will not be reflected in the Small. In practice it is much more important that bounds be computed quickly and accurately than that they should reflect any change in the form of subexpressions. C++ is not garbage collected, but it does provide enough hooks for Smalls to be reference counted, which has been done.

Another case where significant performance improvements can be made is in recalculating exponents of exact numbers. That is, the exponent computed when an exact number is originally allocated may be to large if the expansion of the primitive expression is later found to have leading 0 bigits. In that case the data structure can be transformed by adjusting the exponent:

$$\mathcal{B}^e(\mathbf{expanded}(0, \mathbf{x})) \quad \rightarrow \quad \mathcal{B}^{e-1}\mathbf{x}$$

The base $\mathcal{B}$ is chosen as 2^{13}. This allows most manipulations of small integers to to be carried out on with 32 bit integer arithmetic. (Two bigit accuracy suffices for most manipulations, but in a couple of instances more bits were needed. The worst case was in division where extra bits were required to calculate a properly rounded quotient.)

10 Complexity

There are several ways to ask the question "How complex are operations on exact numbers?" Of course a lot

depends on what exponents are involved. In the the discussion of this section I assume that all the exact numbers involved have the same exponent (and the primitive expressions are not `shift`'s). For multiplication and division exponents are irrelevant to complexity issues.

10.1 Basic Operations

How much slower are these algorithms than ordinary floating point if no more precision is ever required than can be provided by floating point?

This is a somewhat vague question and the answer depends on exactly what operations are done and on low level implementation details. Measurements of the implementation discussed in section 9 yield factors of 50 to 100. That is, hardware floating point is 50 to 100 times faster than that implementation. It is possible that some tuning of the implementation can improve these results.

10.2 Multi-bigit Operation

How much work does it take to expand n bigits of a sum, product or quotient if the subexpressions are fully expanded?

For this case, the algorithms presented here are equivalent to the ordinary naive implementations of multiple precision arithmetic. Addition is linear in n, multiplication and division are quadratic in n.

10.3 Propagation of Demand

Given an expression that combines exact numbers, how many bigits of the exact numbers will have to be expanded for each bigit of the combination that is expanded?

This is in many ways the most interesting question, and it is also the hardest to answer because the algorithms have been deliberately designed to take into account the structure of the combinations. The results of section 6 give some lower bounds. But when expanding $x + y$ the algorithms do not work by performing some predetermined manipulations on x and y and then combining the result. Instead, they perform some manipulations on one of the operands, look at the result and use that to determine what manipulations to do next. So the actual performance can in many cases be much better.

To analyze addition note that the `range` of an expanded primitive expression is (at most) 2. When adding `x` and `y`, it is never necessary to expand them further if $\texttt{range}(\texttt{x}) + \texttt{range}(y) < \mathcal{B}$. Thus it is possible to form a primitive expression for the sum of $\mathcal{B}/2$ exact numbers whose primitive expressions each have a single bigit of expansion without forcing any more expansion. And this can occur however the expression is "shaped". That is it applies to an unbalanced "tower":

$$(x + (x + (x + \ldots)))$$

As well as to more balanced expressions:

$$((\ldots) + (\ldots)) + ((\ldots) + (\ldots))$$

Multiplication is harder to analyze than addition. The techniques for limiting propagation of demand are not as effective as for addition and a multiplicative tower performs signficantly worse than a balanced expression.

References

[1] T. E. Hull, A. Abraham, M. S. Cohen, A. F. X. Curley, C. B. Hall, D. A. Penny, and J. T. M. Sawchuk. Numerical turing. *SIGNUM Newsletter*, 20(3), July 1985.

[2] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[3] Hans-J. Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In *1986 ACM Conference on LISP and Functional Programming*, Cambridge, Mass., 1986.

[4] Hans-J. Boehm. Constructive real interpretation of numerical programs. In *SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, 1987.

[5] Jean Vuillemin. Exact real computer arithmetic with continued fractions. In *1988 ACM Conference on LISP and Functional Programming*, Salt Lake City, Utah, 1988.

[6] Jerry Schwarz. A C++ library for infinite precision floating point. In *C++ Conference*, 1988.

[7] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.