TOWARD AN IDEAL COMPUTER ARITHMETIC

*T.E. Hull and M.S. Cohen**

Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Abstract

A new computer arithmetic is described. Closely related built-in functions are included. A user's point of view is taken, so that the emphasis is on what language features are available to a user. The main new feature is flexible precision control of decimal floating-point arithmetic. It is intended that the language facilities be sufficient for describing numerical processes one might want to implement, while at the same time being simple to use, and implementable in a reasonably efficient manner. Illustrative examples are based on experience with an existing software implementation.

1. Introduction

Our purpose is to describe a new computer arithmetic, for which we have developed prototype implementations (both in hardware [2,3] and software [7]) and which we believe provides significant advantages over what is currently available, especially for anyone interested in numerical computation. We call it CADAC arithmetic because CADAC is the acronym we used for the original hardware implementation (from Clean Arithmetic with Decimal base And Controlled precision).

We take a user's point of view, and so we describe the arithmetic (along with its related built-in functions) in terms of programming language facilities. Our goal is to have language facilities which are near to ideal, in some reasonable sense, from a user's point of view. Above all, the facilities should therefore be *sufficient*, in that they should enable a user to describe any numerical process that the user might wish to implement. But they should otherwise be as *simple* as possible, and they should be implementable in some reasonably *efficient* manner.

The facilities described in this paper are intended to be one example of an arithmetic system which is both sufficient and simple. Our implementations have so far not been particularly efficient, although we believe that reasonable efficiency will eventually be possible (for example, with the development of an appropriate co-processor). However, the question of efficiency is a complicated one, and we will postpone further discussion of it until the last section. In any event, we believe it is a useful exercise to explore what kinds of language facilities would be especially convenient from a user's point of view. Quite a few features of currently available language facilities are a result of decisions which were convenient from the point of view of hardware designers or compiler writers. This of course is not surprising, especially since the users have not, as a whole, made great efforts to agree on what would be ideal.

The main new feature of CADAC arithmetic is precision control. Precision can be declared at any point in a program and, within the scope of that declaration, subsequent floating-point variable declarations, or any operations involving floating-point values, are executed in the declared precision. The precision so specified can be any integer expression, and can be changed dynamically. As will be shown, increasing precision for critical stages of a calculation can often be exactly what is needed to provide a specified accuracy in the final result, or to get around a special difficulty. Being able to increase precision dynamically, so that a problem (such as a system of equations) is solved in higher and higher precision, until some error criterion is satisfied, can sometimes make it possible to solve a problem that would otherwise be virtually impossible.

The arithmetic is decimal, because that is convenient for the user, not only in understanding the arithmetic itself, but also in knowing that input-output is done exactly and program constants are represented exactly. Normally the arithmetic is properly rounded (i.e., to nearest, or to nearest even in case of a tie), but directed roundings are also available. The related built-in functions (including functions for getting and setting exponents, as well as the more usual functions such as those for finding quotients and remainders) are very carefully specified. Other required functions (including the elementary functions for calculating square roots, exponentials, sines, cosines, etc.) can be written in terms of the built-ins, and often take advantage of the precision control capability.

Exception handling is also a part of the new facilities, but this feature is the subject of a

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

* Present address: 141 North Meadow Crescent, Thornhill, Ontario, Canada, L4J 3C4.

separate paper [8]. We do not need to discuss this topic any further in what follows except to specify the circumstances under which exceptions are raised by the various operators we describe. Only three exceptions can be raised by the arithmetic operators, namely *overflow*, *underflow* and *domainerror*.

The next two sections are preliminary to the precise specifications of the arithmetic and the related built-in functions. The purpose of section 2 is to provide an overview of the new language facilities and to illustrate with a few examples. It is not intended that all the details are explained, but rather that a general idea of the new facilities is presented and that some motivation is provided for the specifications that are to follow. Section 3 covers specifications for integers and integer arithmetic. We need to make these precise before presenting the floating-point specifications.

Sections 4, 5 and 6 are then devoted, respectively, to floating-point arithmetic, the related built-in functions, and a brief discussion of other required functions that can be derived from the built-in arithmetic and related functions.

Finally, in section 7, we return to the efficiency question mentioned earlier.

2. Overview and Examples

Our purpose in this section is to give a brief indication of how the new language facilities can be used, especially with regard to precision control, and to thereby provide some motivation for the detailed specifications presented in later sections. The examples are intended to show what a wide variety of situations can arise, some requiring double precision, some only a small increase, some more than double, some in which precision has to be increased repeatedly, and so on.

The examples are written in Numerical Turing [7], which is an extension of Turing [4], a Pascal-like language that has been widely used at the University of Toronto since 1983.

Floating-point variables (which we will often refer to as float variables) are declared with the keyword *real*, and this keyword may optionally be followed by an integer expression in parentheses, as in

```
var a : real(16)
var b : array 1..100 of real(10)
var c : real(2*i + 2)
```

The integer expression is the precision. If the precision is not specified (and it normally is not), the precision defaults to the current precision (which is explained below).

The precision of float operations is declared separately from the precision of the operands. Such precision declarations are of the form

```
precision intExpn
```

where *intExpn* is any integer expression.

The execution of any operation governed by such a declaration is carried out in the precision specified. The scope of such a declaration extends from just after the declaration to the end of the construct in which the declaration is made. Unless explicitly overruled by a subsequent precision declaration, it covers all float operations appearing in that scope, including any that appear in nested subconstructs (begin-end, loops, etc., and the interiors of functions or procedures *invoked* within that scope).

This rule determines the current precision at any particular point in a program, except when no precision declaration has been made, in which case the current precision defaults to 10. The current precision can be determined whenever required – it is the value returned by the built-in function *currentprecision*.

As a first example, suppose that we wish to generate pseudo-random numbers uniformly distributed in the interval [0,1). If we decide to carry 10 digits after the decimal place, so that the numbers are of the form

$$0 \cdot d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10},$$

where the d_i's are decimal digits, we can choose a multiplier m and an additive constant c so that the generator suggested by the congruence

$$r_{k+1} \equiv (m \times r_k + c) \bmod 1$$

will sequence through all numbers of the above form before repeating. (See Knuth [9, pp. 155-156] for criteria to be satisfied by m and c.)

Then, once r has been declared to be of precision 10, as in

```
var r : real(10)
```

and initialized to some value of the above form in the interval [0,1), subsequent values of r can be obtained by executing

```
begin
    precision 20
    r := (m*r + c) mod 1.0
end
```

In practice we would incorporate this generator in a procedure, which could then be used with any of several sequences of random numbers (differing only in their initializations), but we want to avoid introducing more language constructs here than are needed to illustrate the basic ideas about precision control.

As a second example, we consider the dot product of two vectors, say a and b, which is of the form

$$a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

The effect of rounding errors can be reduced by accumulating this dot product in higher precision. Suppose the accumulation is done in extended precision, say xp, while the current precision is denoted by cp. Then, if the additions are performed

from left to right, the usual backward error analysis shows that the final result is exactly

$$(a_1 b_1 (1+\varepsilon_{xp})^n + a_2 b_2 (1+\varepsilon_{xp})^n + \cdots + a_i b_i (1+\varepsilon_{xp})^{n+2-i} + \cdots + a_n b_n (1+\varepsilon_{xp})^2)(1+\varepsilon_{cp}),$$

where the ε_{xp}'s are relative roundoff errors in extended precision (the ε_{xp}'s are not necessarily the same), and therefore satisfy $|\varepsilon_{xp}| \leq .5 \times 10^{-xp+1}$, while ε_{cp} is a relative roundoff error in current precision and therefore satisfies $|\varepsilon_{cp}| \leq .5 \times 10^{-cp+1}$.

The point is that the final result is the exact dotproduct of two vectors that differ very little from the original two vectors. In fact, their individual components do not differ by more than the factor

$$(1+\varepsilon_{xp})^n (1+\varepsilon_{cp}) \approx 1 + n\,\varepsilon_{xp} + \varepsilon_{cp}.$$

For the extended precision contribution to the error in this expression to be roughly negligible compared to the maximum that can be expected from the current precision contribution – say no more than 10% of the latter – we can choose

$$n \times .5 \times 10^{-xp+1} \leq .1 \times .5 \times 10^{-cp+1},$$

which reduces to

$$xp \geq cp + \log_{10}(n) + 1.$$

This enables us to choose an appropriate extended precision for accumulating dotproducts, with the help of the *getexp* function, since, as we later define the built-in function *getexp* for getting the exponent of a floating-point number, getexp(n) is an upper bound for $\log_{10}(n)$. (The integer n is converted to a normalized floating-point form before the *getexp* function is evaluated.) Thus, once the vectors a and b have been declared and the values of their components have been determined, their dot product can be accurately evaluated with

```
var dotproduct : real
begin
    precision currentprecision + getexp(n) + 1
    var temp : real := 0.0
    for i : 1..n
        temp := temp + a(i)*b(i)
    end for
    dotproduct := temp
end
```

Here the precision of *dotproduct* is the precision that is current when its declaration is made. (We are assuming that a and b are also in this precision.) The precision of *temp* is the extended precision, and *temp* is initialized to the value 0.0. All the arithmetic is done in extended precision, so that the final value of *temp* is the result of accumulating the dot product in extended precision. Then this final value of *temp* is rounded to the precision of *dotproduct*, before being assigned to *dotproduct*.

With other programming languages, dot products are usually not accumulated in higher precision.* If they are, the accumulation would normally be done in double precision, but that may be impossible, or at least quite awkward to do. In any event, the analysis given above shows that only a rather modest increase in precision is needed to produce a satisfactory increase in accuracy, and the facilities presented in this paper enable us to describe the process in a straightforward manner.

Another example occurs in the evaluation of the exponential function. One program developed for this purpose [6] is based on the identity

$$(e^{x/k})^k = e^x,$$

where k is chosen so that $|x/k| < 1$. The program uses a polynomial approximation to $e^{x/k}$. This polynomial is evaluated in higher precision, and then raised to the power k in the same higher precision, before being rounded down to the current precision of the environment in which the function is invoked. To return a value which is guaranteed to differ from the true value of e^x by less than 1 unit in the last place (which is required of all the elementary functions in Numerical Turing), the required higher precision turns out to be

```
currentprecision + max(getexp(x),0) + 2
```

where *currentprecision* is the precision of the environment in which exp(x) is invoked.

As a fourth example, we consider a process involving Newton's method. Newton's method is often the basis for solving non-linear equations, and it is well known that the iterations can usually be started in quite low precision, and then approximately doubled with each iteration. For example, to calculate an approximation to the square root of f, it can be shown that precisions $2^i + 2$, for $i = 1,2,\cdots$, up to *currentprecision* + 2, are sufficient to ensure an approximation, after the final rounding to the current precision of the calling environment, that is in error by less than 1 unit in the last place, provided an initial approximation within 10% has been obtained [5]. If this initial value is assigned to *approx*, the iterations can be implemented as in the following [5, p. 231]:

```
var p := 3
const maxp := currentprecision + 2
loop
    p := min(2*p - 2, maxp)
            % p = 4, 6, 10, ... maxp
    precision p
    approx := .5*(approx + f/approx)
    exit when p = maxp
end loop
```

Another example in which precision has to be changed dynamically is in a program for solving a system of linear equations to within a prescribed accuracy. Such a program can solve the system in a

* With language extensions that support ACRITH, dot products are accumulated exactly (e.g., see [10]).

particular precision, and then calculate an estimate or a bound on the error in the computed solution. If the estimate or bound is less than what was prescribed, the program is finished. If not, it can simply redo the solution in higher precision, and repeat the process if necessary until the estimate or bound is small enough. Meeting such a requirement without such a flexible precision control would be virtually impossible.

In each of the examples described so far, precision has been controlled in order to provide sufficient accuracy. However, in CADAC arithmetic, the exponent range changes with the precision (if p is the precision, the exponent must be in $[-10^p, 10^p]$), and we sometimes increase the precision in order to increase the exponent range as well as increasing the accuracy.

One such situation occurs in a procedure for solving quadratic equations. If the equation is of the form

$$ax^2 + bx + c = 0,$$

the well-known difficulty is in evaluating the discriminant $b^2 - 4ac$. The problems to be avoided are the possible overflow or underflow. With CADAC arithmetic, the solution is very straightforward. It turns out to be sufficient to evaluate this expression in precision

```
2*currentprecision
        + ceil(currentprecision/5)
```

(Here *ceil(u)* is the smallest integer $\geq u$.) We need the detailed specifications in sections 3 and 4 to make this precise and to prove that it is correct. For our purposes here, it is enough to note that we need to roughly double the current precision in order to obtain sufficient accuracy *and*, because the exponent range is then roughly doubled as well, to avoid overflow and underflow in the evaluation of b^2 and $4ac$. But we have to increase the precision somewhat more to also avoid any possibility of underflow in the subtraction. It can be shown that the precision given above is the smallest possible precision with all these properties. The resulting approximation to $b^2 - 4ac$ will suffer at most one rounding error in the final subtraction.

As a final example, consider evaluating the Euclidean norm of the vector a, namely

$$(a_1^2 + a_2^2 + \cdots + a_n^2)^{1/2}.$$

Here it is easily shown that no overflow or underflow can occur if the precision is

```
2*currentprecision + getexp(n) div 10 + 1.
```

Of course the rounding of the final result back to the calling precision can still cause an overflow, but this is as it should be – it means that the Euclidean norm of a cannot be represented in the precision of a. The high precision has ensured that no unnecessary intermediate overflow or underflow can occur.

In practice it would be better, if possible, to avoid using such high precision for calculating every Euclidean norm. Such high accuracy is generally not needed, and the chances of intermediate overflow or underflow are very small. A better strategy is to leave the precision unchanged, at the current precision, but to see that the calculations are redone in higher precision if any overflow or underflow should occur in the course of the calculation in current precision. This strategy is also easily implemented, but requires the use of exception handlers. This particular example is discussed in [8]. The point to be made with this example is that precision control can sometimes be helpful in handling exceptions.

3. Integer Arithmetic

It is convenient to consider integer arithmetic before getting into the details of floating-point arithmetic. The two arithmetics have some points of contact, so that you cannot specify one completely without the other. We do not have anything very unusual to say about integer arithmetic and we will therefore only sketch the specifications rather briefly, but there are a few points that need to be attended to, especially to make sure that the arithmetic is not only completely specified but also is as simple as is reasonably possible.

We begin by requiring a function, *maxint*, which returns a positive integer value, and we require all legal integer values to be in the closed interval [-maxint, maxint]. By not allowing a function *minint*, which can return a value that might not be equal to *-maxint*, we avoid the possibility of overflow with expressions of the form $-i$ and abs(i), where i is an integer variable. With the current implementation of Numerical Turing, *maxint* returns the value of $2^{31} - 1$.

The prefix operators + and - and the infix operators +, - and * are defined as one would expect. Overflow can occur with any one of the three infix operators.

There is no division operator for integers. If an expression of the form i / j occurs in a program, where i and j are integers (or integer expressions), the values of i and j are converted to floating-point form before the division is carried out (as described in the next section).

The power operator ** is also defined as one would expect, as long as, in expressions of the form i ** j, where i and j are integer expressions, it is not the case that $i = j = 0$ or that $j < 0$. Of course an overflow exception can occur. If $i = j = 0$, a domain error exception occurs. If $j < 0$, the value of i is converted to floating-point form before the power operation is carried out; this case is described in the next section.

Two infix operators, denoted by *div* and *mod*, provide truncated division and remainder, respectively. A domain error exception occurs if $j = 0$ in either i div j or i mod j. The two operators satisfy the relation

$$i \bmod j = i - (i \operatorname{div} j) * j$$

as long as $j \neq 0$.

To complete the specification of expressions involving integers, we would now need to specify operator priorities, the order of evaluation when priorities are equal (as, for example, in $i + j + k$), the comparison operators, and so on. But this paper is mainly concerned with the basic arithmetic and closely related operators, and we will therefore not go further into a discussion of expressions in general.

4. Floating-Point Arithmetic

We require a function, *maxprecision*, which returns a positive integer value. We also require $10 \times maxprecision \leq maxint$. This last requirement is a very modest one, but, as we will see later, it helps make the system simple, especially in the definition of the *getexp* function. (In the current implementation of Numerical Turing *maxprecision* is 200.)

Any legal floating-point value can be represented either by zero, or as a normalized, p-digit, decimal, floating-point number with an exponent in the closed interval $[-10p, 10p]$. A floating-point number is *normalized* if its significand is in the half-closed interval $[.1, 1)$. The precision p must be in the closed interval $[1, maxprecision]$.

A particular floating-point value can of course be represented in different ways. For example, the value 12.34 can be represented as 1.234×10^1 or 1234×10^{-2} or $.1234 \times 10^2$, but only the last of these is *normalized*, according to the convention we use for normalization.

The keyword *real* is used in the declaration of floating-point variables, as in

```
var a, b : real
```
or
```
var x, y : real(p)
```

In the first case, which is the usual case, a and b will have the current precision of the environment in which they are declared. In the second case, x and y will have precision equal to the value of p, where p is any integer expression (provided that value is in $[1, maxprecision]$).

Within the scope of any precision declaration, all expressions will be in the specified precision or, if not already, they will be coerced into that precision. In particular, any constant or variable that appears as an expression or part of an expression will be rounded to a lower precision, if necessary, before it is used. (Rounding means "proper rounding", i.e., to nearest, or to nearest even in case of a tie.) For example, in the following

```
precision 2
var y : real
precision 6
var x : real
x := 3.14159
precision 4
y := x*x + 1.2
```

the value of each x in the last line will be rounded to 3.142 before the multiplication is carried out, in precision 4, followed by the addition of 1.2, also in precision 4. Then the resulting value is rounded to precision 2 before being assigned to y.

Neither overflow nor underflow can occur in this example, but it should be noted that such roundings could, in general, cause overflow or underflow to occur, because the lower precision has a smaller exponent range. However, for the purposes of this paper, the more important point to notice is that, for the specification of any operator (such as the multiply operator in this case), we can assume that any arguments are always in the same precision and that the result of the operation is also to be delivered in that precision. A final assignment may involve another coercion (such as the assignment to y in this case), and could therefore cause overflow or underflow to occur (although not in this particular case).

The prefix operators + and - in floating-point arithmetic are as one would expect. However, some care is needed in specifying the infix operators +, -, * and /. Our specifications are given in Figure 1.

```
if op is / and b = 0.0 then
    result is domain error exception
else
    m := mathematically exact value
            corresponding to a op b
    if m = 0.0 then
        result is 0.0
    else
        r := m properly rounded to
                p digits and normalized
        if exponent of r > 10p then
            result is overflow exception
        elsif exponent of r < -10p then
            result is underflow exception
        else
            result is r
        end if
    end if
end if
```

Figure 1. Specification of a *op* b where *op* is one of +, -, * and /. It is assumed that a and b are in precision p, where p is in $[1, maxprecision]$, and that the result is to be delivered in precision p. (Note that the exponent range in precision p is $[-10p, 10p]$, and that "properly rounded" means "to nearest, or nearest even in case of a tie".)

The infix power operator ** was defined in the previous section for the cases when both arguments are integers. One of those cases required conversion to one of the cases that are now about to be considered.

When the first argument has a floating-point value and the second (the power) is an integer, we use the notation $a^{**}n$, and distinguish various cases in the following way

```
if both a and n are zero then
    result is domain error exception
elsif only n is zero then
    result is 1.0
elsif n > 0 then
    a**n is (...(((a)*a)*a)...)*a
                % n-1 multiplications
or, with r := 1/a,
    a**(-n) is (...(((r)*r)*r)...)*r
                % n-1 multiplications
end if
```

(The result is an overflow or underflow exception if any multiplication, or the division to evaluate r, causes overflow or underflow, respectively, to occur.) Approximations for large values of n can of course be obtained with fewer multiplications but derived functions can be developed to exploit such possibilities. We prefer the more straightforward specification given here because of its simplicity. Such derived functions are not more accurate. In fact, for even larger values of n, approximations based on the use of logarithm and exponential functions are both more accurate and more efficient. This is the approach used when both arguments of the power operator are floating-point values, and is therefore included in section 5 along with the elementary and other derived functions.

The quotient and remainder operators, *div* and *mod*, were specified with integer arguments in the preceding section. If only one of the arguments is an integer, it is converted to floating-point form before the operation is carried out. We therefore have left only to specify these operations when both arguments are in floating-point form, and this is done in Figures 2 and 3. (Note that there can be no rounding error with *mod*.)

Directed roundings can be very useful in scientific computing. The "round up" operations are specified in Figure 4 and the "round down" operations are specified in Figure 5. The "round towards zero" operations are closely related (like "round up" when the result is negative, but like "round down" when the result is positive), but, in any event, do not appear to be as useful, so they are not given here in any more detail. The present implementation is in terms of functions (e.g., addru(a,b) produces the "rounded up" sum of a and b), but we plan to change to infix operators, at least for "round up" and "round down", probably something like a +' b for addru(a,b) and a '+ b for addrd(a,b), and similarly for -, * and /.

As with the integers, further specification of expressions involving floating-point numbers would require a description of operator priorities, relational operators, and so on, but our main interest here is in the basic arithmetic operations themselves, along with the closely related *div*, *mod*, directed roundings, and some cases related to the power operator.

```
if b = 0.0 then
    result is domain error exception
else
    m := the mathematically exact value
            of a ÷ b rounded to the nearest
            integer in the direction of zero
    if |m| > maxint then
        result is overflow exception
    else
        result is m
    end if
end if
```

Figure 2. Specification of a div b, where a and b are in floating-point form.

```
if b = 0.0 then
    result is domain error exception
else
    m := the mathematically exact value
            of a ÷ b rounded to the nearest
            integer in the direction of zero
    r := the mathematically exact value of a-bm
    if r = 0.0 then
        result is 0.0
    else
        nr := normalized form of r
        if exponent of nr < -10p then
            result is underflow exception
        else
            result is nr
        end if
    end if
end if
```

Figure 3. Specification of a mod b, where a and b are in floating-point form, and p is the precision.

5. Other Built-in Functions

The *abs* function is specified as one would expect. The type of its result is the same as the type of its argument. Because of the way in which integers and floating-point numbers have been specified (especially that integers must be in the interval [-maxint, maxint]), no exception can occur with *abs*.

The *min* and *max* functions must each have exactly two arguments, and the type of the result in each case is the same as the type of its arguments, if the arguments are of the same type. If the arguments are of different types, the integer argument is converted to floating-point form before the minimum or maximum is determined.

```
if the ''round up'' operation is /
                and b = 0.0 then
    result is domain error exception
else
    m := the mathematically exact value
                corresponding to a op b
    if m = 0.0 then
        result is 0.0
    elsif m > (1-10^-p) x 10^10p  then
        result is overflow exception
    else
        result is the algebraically smallest
                representable number ≥ m
    end if
end if
```

Figure 4. Specification of a op b for the "round up" versions of +, -, * and / in precision p. A representable number in precision p is zero, or its p-digit normalized form has an exponent in [-10p, 10p]. The largest such number has the value $(1-10^{-p}) \times 10^{10p}$.

```
if the ''round down'' operation is /
                and b = 0.0 then
    result is domain error exception
else
    m := the mathematically exact value
                corresponding to a op b
    if m = 0.0 then
        result is 0.0
    elsif m < -(1-10^-p) x 10^10p  then
        result is overflow exception
    else
        result is the algebraically largest
                representable number ≤ m
    end if
end if
```

Figure 5. Specification of a op b for the "round down" versions of +, -, * and / in precision p. A representable number is as defined in the caption for Figure 4. The algebraically smallest such number has the value $-(1-10^{-p}) \times 10^{10p}$.

Four functions are available for conversion. The *floor*, *ceil* and *round* functions convert from float to integer, to "greatest lower bound", "least upper bound" and "properly rounded", respectively. In each case an overflow exception can occur. The fourth function, *intreal*, converts integer to float. The resulting float value will be in the current precision, and, depending on this precision, a rounding error may have occurred – but, if so, the result will be properly rounded. In principle, it is also possible for overflow to occur, but this cannot happen with the present implementation, since the largest integer value $(2^{31}-1)$ is less than the largest representable float, even in precision 1 (namely $.9 \times 10^{10}$). The *intreal* function is invoked implicitly whenever an integer value appears in an expression where a float is expected, such as with a divide operator, or in one case with the power operator, or with assignment to a float variable.

Conversion between float values of different precisions is done implicitly when a value in one precision is used in an expression when the current precision is different, or when assignment is made to a variable whose precision differs from the current precision. If the conversion is from a higher precision to a lower one, overflow or underflow may occur.

Two functions, *getexp* and *setexp*, have proven to be extremely useful. The result of getexp(a), where a is a float, is the integer exponent of a if $a \neq 0.0$ (and a is in normalized form), but 0 if $a = 0.0$. The result of setexp(a,n), where n is any integer, or integer expression, is the normalized float value of a, except that a's exponent has been replaced by the value of n if $a \neq 0.0$, or 0.0 if $a = 0.0$. Overflow or underflow occurs if $n > 10p$ or $< -10p$, respectively, where p is the current precision.

We also need a function, *precisionof*, for determining the precision of a variable. This function is normally used in the body of a function or procedure to determine the precision of one of the parameters being passed to that function or procedure. It is necessary to allow (read-only) *reference* parameters, even with functions, to be able to take advantage of this possibility. Such parameters are allowed in Numerical Turing, but not in Turing. There are other situations in which it is convenient to use reference parameters, and one important one will be indicated in the next section.

6. Derived Functions

Two properties of "ideal" language facilities which we identified in the introduction to this paper were that the facilities be *sufficient* for describing the processes we want to implement, and that they also be *simple*. We hope that the examples mentioned so far, along with the specifications described in the preceding three sections, have provided good support for our claim that the facilities described in this paper do satisfy these criteria reasonably well.

In further support of this claim we will mention a few of the functions that have been developed with these facilities. To begin with, the elementary functions for Numerical Turing were written in Numerical Turing. They include sqrt, ln, exp, sin, cos, arctan and the power function, and are based on the thesis by Abrham [1]. See also [5,6]. In each case they deliver an approximation that is in error by less than one unit in the last place, in the precision of the calling environment, over an appropriate range of the argument. The programs are relatively straightforward, mainly because it is possible to change precision when necessary to attain the desired accuracy, but also because *getexp* and *setexp* are available.

A function for determining an approximation to π, to within about one unit in the last place, in the precision of the calling environment, is also available.

A number of functions have been developed for working with the digits of a number, such as for determining the next larger number, for determining the size of a unit in the last place, and so on. One function that has proven to be quite useful is *realr*, where the value of realr(a, p) is the same as the value of a, except that a has been rounded to p digits. (Here a must be a reference parameter.)

We have also experimented successfully with functions for doing complex arithmetic and interval arithmetic. But, eventually, we plan that at least complex arithmetic will use the same operators as are used with float arithmetic, and perhaps interval arithmetic will use these as well.

7. Concluding Remarks

The third criterion for "ideal" language facilities which we identified in the introduction was that the facilities be implementable in some reasonably *efficient* manner.

Our hardware implementation [2,3] has recently been retired. Our software implementation has been improved over what was described in [7], but a floating-point intensive subprogram, such as a subprogram for approximating the sine function, is still quite slow, as one would expect.

For efficiency, we plan to develop a co-processor to support CADAC arithmetic. The actual add and multiply times for a particular precision, say 14 or 15 decimal digits, will be somewhat longer than for a corresponding fixed precision implementation – perhaps about one and one half times as long, depending on design details.

But these times are only one factor in determining the overall efficiency of a system. We must also, for example, consider memory access times, and the extent to which these times might overlap with add and multiply times. And we must consider the advantages in being able to control the precision, not just to get more accuracy at critical points in a program, but also for ease of programming (which will sometimes thereby be more efficient), and proving programs correct. In some cases, one can even solve problems which would otherwise be practically impossible to solve.

There can of course be no single measure of efficiency for all purposes. But, for some purposes, we believe a reasonably efficient implementation of CADAC arithmetic is possible, and would be capable of supporting language facilities which are very convenient (sufficient and simple) for numerical computing.

Bibliography

1. Abrham, A. Variable Precision Elementary Functions. M.Sc. thesis, Department of Computer Science, University of Toronto, Toronto, 1985.
2. Cohen, M.S., Hamacher, V.C. and Hull, T.E. CADAC: An Arithmetic Unit for Clean Decimal Arithmetic and Controlled Precision. *Proceedings 5th Symposium on Computer Arithmetic* (IEEE Computer Society, Ann Arbor, Michigan, 1981), 106-112.
3. Cohen, M.S., Hull, T.E. and Hamacher, V.C. CADAC: A Controlled-Precision Decimal Arithmetic Unit, *IEEE Transactions on Computers*, vol. C-32, 4 (1983), 370-377.
4. Holt, R.C. and Cordy, J.R. The Turing Language Report. Technical Report CSRI-153, Department of Computer Science, University of Toronto (revised July 1985). (An earlier version of this report also appears as an appendix in a text by Holt, R.C. and Hume, J.N.P. *An Introduction to Computer Science Using the Turing Programming Language*. Reston, Reston, Va. (1984).)
5. Hull, T.E. and Abrham, A. Properly Rounded Variable Precision Square Root. *ACM Trans. Math. Softw. 11*, 3 (Sept. 1985), 229-237.
6. Hull, T.E. and Abrham, A. Variable Precision Exponential Function. *ACM Trans. Math. Softw. 12*, 2 (June 1986), to appear.
7. Hull, T.E., Abrham, A., Cohen, M.S., Curley, A.F.X., Hall, C.B., Penny, D.A. and Sawchuk, J.T.M. Numerical Turing. *ACM SIGNUM Newsletter 20*, 3 (July 1985), 26-34.
8. Hull, T.E., Cohen, M.S., Sawchuk, J.T.M. and Wortman, D.B. Exception Handling in Scientific Computing. In preparation.
9. Knuth, D.E. The Art of Computer Programming, vol.2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass. (1969).
10. Kulisch, Ulrich W. and Miranker, Willard L. [Eds.]. A New Approach to Scientific Computation. Academic Press, New York (1983).