ISSN 0105-8517

PAL — A Language for Multiple-Precision Algorithms

Torben Hagerup

DAIMI PB-165
August 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

PB-165

T. Hagerup: PAL — A Language for Multiple-Precision Algorithms

TRYK: DAIMI/RECAU

PAL - A Language for Multiple-Precision Algorithms.

Abstract.

Traditional notations (programming languages) are seldom adequate for the description of multiple-precision computations. This is because there is no way to specify how accurately the arithmetic operations should be carried out. This paper presents a special-purpose programming language which allows multiple-precision algorithms to be stated accurately and elegantly.

1. Introduction.

A number of people have studied algorithms which involve multiple-precision real numbers. Brent [1] and Olver [4] both aim at providing unrestricted algorithms for elementary functions, i.e. algorithms that can produce arbitrarily close approximations to values of the function in question. Such algorithms are often stated in a way that is unsatisfactory for a number of reasons.

Suppose for instance that we want to describe an algorithm which uses a well-known special case of the Newton-Raphson method to compute square roots to any specified precision. It turns out that the iteration formula

$$y_{n+1} = y_n + \frac{x - y_n^2}{2y_n}$$

is more efficient than the commonly used

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right) .$$

Ignoring the question of how to find a suitable initial value which is not important here, a traditional description might consist of the Pascal function

```
01  FUNCTION mp_sqrt(x,eps:real):real;
02
03    VAR y,old_y:real;
04
05  BEGIN
06    y:=initial_value(x);
07    REPEAT
08      old_y:=y;
09      y:=y+(x-sqr(y))/(2*y);
10    UNTIL abs(y-old_y)<=eps;
11    mp_sqrt:=y;
12  END;
```

Algorithm 1.

together with the following explanatory remarks:

Compute in single precision until y and old_y agree to single precision. Perform the next iteration in double precision, and in general double the number of words in each

of the following steps (this is because the Newton-Raphson iteration is quadratically convergent).

This type of description suffers from severe drawbacks:

- It is unfortunate that a formal description like the program text need be "interpreted" by additional informal comments.

- It is stated how accurately each iterand should be computed. However, nothing is said about how accurately each primitive operation such as an addition must be carried out in order to achieve the required overall accuracy. In other words, the description does not go below statement level. For instance, it turns out that even if we want a double-precision result in line 09, the division need only be carried out to single precision. Such information can of course also be given informally, but the description tends to become increasingly clumsy and ambiguous.

- The description makes reference to the number of computer words. First of all, this is a concept which is foreign to the problem at hand as well as to the solution method, i.e. the Newton-Raphson method. It is an unnecessary complication to worry about the size and number of computer words, and it introduces an undesirable potential machine dependence.

  Secondly, computer words by definition come in integral numbers. Although it is quite likely that addition, multiplication and perhaps even division of multiple-precision numbers are "word-oriented", i.e. always give an integer number of correct result words, there is no reason why this should also hold for functions like 'mp_sqrt' (which may in turn be used as a primitive in other algorithms). At any rate, measuring accuracy by the number of words seems an unnatural discretization, even for addition and multiplication. Of course, one can attach a meaning to a statement like "y is computed with a precision of 2.7 words" by means of a suitable smooth definition. But phrases of this kind are not likely to contribute to the clarity of the description. Essentially the same problems arise if accuracy is measured in bits or (decimal) digits.

  Thirdly, "a and b agree to single precision" seems to imply that the first n bits of a and b are identical, where n is the number of bits per word. But what is meant is more often that the relative deviation between a and b is smaller than 2^{-n}. Even if the latter condition holds, it is possible that no corresponding bits of a and b are identical.

  Finally, it is at least conceivable that the representation is such that it does not make sense to talk about "single-precision", "double-precision" etc.

The purpose of this paper is to present a programming language PAL that attempts to overcome most of the difficulties. A detailed description of PAL is outside the scope of the paper. It merely aims at exposing and discussing the most important ideas behind the design of the language.

PAL is a special-purpose language intended for use in the field of multiple-precision numerical computations. It is a simple, Pascal-based language designed to be easy to compile and to allow of an efficient implementation. An implementation, described in [3], actually exists at the University of Århus. It has been used successfully in a number of projects; in particular, a collection of unrestricted routines for many basic mathematical functions has been written in PAL [2].

2. Language description.

2.1 Language overview.

PAL borrows much of its syntax and semantics from Pascal. The most significant differences have to do with expressions and arithmetic operators. More traditional aspects of the language will not be described in any great detail.

PAL supports only one basic type which is called 'PAL real'. The set of its values is a subset of the real numbers and in particular includes the integers. A PAL real has no accessible internal structure. It may in fact be represented as an integer, stored in a number of computer words, plus a scale factor, but this is completely transparent to the programmer.

I shall not always in the following distinguish between PAL reals and their (real) values.

PAL reals can be manipulated by various built-in operators. Each of these is an approximation to an "ideal", "mathematical" operator which I call the associated ideal operator. Correspondingly, I shall talk about the ideal value as opposed to the value actually produced by an operator.

As a radical departure from Pascal, an arithmetic operator in general takes, in addition to its usual arguments, an extra argument specifying the precision with which the operation must be carried out. There is not, as in

most traditional languages, a default precision which is used in all real operations. As an example, the assignment to z of the sum of the values of x and y takes the form

```
z:=sum(x,y,f)          ,          (1)
```

for some suitably chosen f. 'sum' is the predefined addition operator, and x, y and f may be arbitrary PAL expressions. The meaning of (1) is that the implementation must guarantee that the value assigned to z deviates from the ideal value (i.e., from $x+y$) by less than the value of f (which must be ≥ 0). In other words, f specifies the maximum allowable absolute error.

Similar functions exist for subtraction, multiplication etc. I call these F operators ("F" symbolizing the explicit error bound).

The case with $f = 0$ occurs frequently enough to merit a special shorthand. This gives rise to the exact operators which may be thought of as implementations of the ideal operators. Most of the exact operators have the same syntactic representation as their counterparts in Pascal and other languages. Thus

```
z:=x+y
```

is exactly equivalent to

```
z:=sum(x,y,0)          .
```

A final group of operators consists of the APC operators (automatic precision control operators). These look syntactically like the exact operators but are, from the point of view of precision, similar to ordinary Pascal (or hardware) real operators. They provide an easy-to-specify and efficient way to carry out computations where precision is not essential. The actual precision of the APC operators is implementation-dependent. However, even the APC operators must adhere to a set of axioms (see section 2.4).

The ambiguity between exact and APC operators is resolved by means of a number associated with each PAL expression called its bias. The bias of an expression is a static (compile-time) attribute which may be -1, 0 or 1. The principal (outermost) operator of the expression is said to have the same bias. Each operator symbol which would otherwise have been ambiguous denotes an exact operator if its bias is zero, and an APC operator otherwise.

The bias also acts as a rounding rule. If a predefined operator has a non-zero bias, the error resulting from an application of the operator is guaranteed to be one-sided:

If the bias is 1, the operator will produce a result which is $\geqq$ the ideal value. If the bias is -1, the opposite inequality holds. The bias of user-defined operators (functions) is ignored.

The bias of an expression may be set explicitly by the programmer, or it may be determined by default rules.

2.2 F operators.

The properties of the arithmetic F operators may be expressed succinctly by means of the following notational device:

When x is a real number, $\hat{x}$ denotes a real number y with

$$|y| \leqq |x| \quad .$$

For instance, if $x \geqq 0$,

$$\ln(1+x) = \hat{x} \quad . \qquad (2)$$

When $\hat{x}$ appears on the right-hand side, as in (2), it is existentially quantified ("there exists a y with $|y| \leqq |x|$ so that $\ln(1+x) = y$"). Appearing on the left-hand side, it is universally quantified. Different occurrences of $\hat{x}$ are not correlated in any way (i.e. each may denote a different value, even within the same expression).

Let us further denote by V(x) the value of a PAL expression x. The arithmetic operators can now be defined by the table below in which each operator is shown with example operands.

Operator	Computes
sum(x,y,f)	$V(x)+V(y)+\hat{V(f)}$
dif(x,y,f)	$V(x)-V(y)+\hat{V(f)}$
prod(x,y,f)	$V(x)V(y)+\hat{V(f)}$
quot(x,y,f)	$V(x)/V(y)+\hat{V(f)}$
copy(x,f)	$V(x)+\hat{V(f)}$

Table 1. Arithmetic F operators.

2.3 Exact operators.

The most important exact operators are given in the table below. Notice that there are exact operators which have no corresponding F operator. On the other hand, there is no exact division operator.

Operator	Computes
x+y	V(x)+V(y)
x-y	V(x)-V(y)
x*y	V(x)V(y)
+x	V(x)
-x	-V(x)
x^n	V(x)^V(n)
m DIV n	V(m) DIV V(n)
m MOD n	V(m) MOD V(n)
abs(x)	|V(x)|
ent(x)	⌊V(x)⌋
e(x,n)	V(x)·(10^V(n))
expo(x)	⌊log10|V(x)|⌋

Table 2. Exact operators.

All the arguments shown as m or n must evaluate to integer values.

2.4 APC operators.

All APC operators are defined relative to a positive number called the APC delta and denoted by Δ . Δ is an implementation-dependent constant which must be <1. The implementation at Århus uses

$$\Delta = 10^{-8} \quad .$$

The derived relation $\sim$ is useful in describing the APC operators:

$$x \sim y \iff |x-y| \leq \Delta \cdot \min\{|x|, |y|\} \quad .$$

When $x \sim y$, I shall say that x and y are APC close.

Most arithmetic APC operators can be characterized by saying that the value returned by the APC operator is APC close to the ideal value. Exceptions concern addition and subtraction.

Operation	Computes a number which is APC close to
x+y	see below
x-y	see below
x*y	V(x)V(y)
x/y	V(x)/V(y)
-x	-V(x)
x^n	V(x)^V(n)
abs(x)	|V(x)|
e(x,n)	V(x)·(10^V(n))

Table 3. Arithmetic APC operators.

Again, arguments shown as n must evaluate to integer values. Addition and subtraction obey the following axioms:

$$V(x+y) \sim V(x)+V(y) \qquad \text{if } V(x)V(y) \geqq 0$$

$$|V(x+y)-(V(x)+V(y))| \leqq \Delta \cdot \max\{|V(x)|,|V(y)|\}$$

$$V(x-y) \sim V(x)-V(y) \qquad \text{if } V(x)V(y) \leqq 0$$

$$|V(x-y)-(V(x)-V(y))| \leqq \Delta \cdot \max\{|V(x)|,|V(y)|\}$$

2.5 Relational and logical operators.

PAL also contains relational and logical operators. Because of the lack of different types, truth values must be represented by real numbers. The following correspondence is used:

0 ~ FALSE
1 ~ TRUE .

In accordance with this convention, relational operators always return one of the values 0 and 1, and the expression occurring in statements such as

IF <expression> THEN <statement$_1$> ELSE <statement$_2$>

is interpreted as FALSE (<statement$_2$> executed) exactly if it evaluates to 0. I shall not in the following distinguish between truth values and their real representations.

Logical operators (AND, OR, NOT) are always exact. Relational operators have F, exact and APC variants. However, the error bound is redefined to be a measure of the permitted "fuzzyness" of the operators. As an example, consider the three operators corresponding to the exact relation "less than":

The F operator is called 'lt' and is defined by

$$V(lt(x,y,f)) = \begin{cases} \text{if } |V(x)-V(y)| \geqq V(f) \\ \text{then (if } V(x)<V(y) \text{ then 1 else 0)} \\ \text{else (either 0 or 1)} \end{cases}$$

In other words, the operator is guaranteed to return the ideal result if $|V(x)-V(y)| \geqq V(f)$, but may return either value if $|V(x)-V(y)|<V(f)$.

The exact operator is written '<', and 'x<y' is equivalent to 'lt(x,y,0)'.

The APC operator is also written '<' and defined by

$$V(x<y) = \begin{cases} \text{if } |V(x)-V(y)| \geqq \Delta \cdot \min\{|V(x)|,|V(y)|\} \\ \text{then (if } V(x)<V(y) \text{ then 1 else 0)} \\ \text{else (either 0 or 1)} \end{cases}$$

The APC operator hence returns the ideal value if its arguments are not APC close.

Note that a non-zero bias has the same effect as for arithmetic operators. Thus a relational operator with bias 1 may return either the ideal result or 1, but not 0 unless the ideal result is 0.

2.6 Bias.

The PAL programmer has full control over the bias of each expression occurring in his program. A given bias is specified by prefixing the expression in question by one of the monadic operator symbols '<', '=' and '>', according to the table below

Bias	Operator
-1	<
0	=
1	>

Table 4. Bias operators.

E.g. in

```
>(x+y)
```

,

the expression 'x+y' has bias 1.

No computation is associated with the bias operators. They simply serve as compiler directives.

When an expression has not explicitly been given any bias, its bias is determined by the following default rules:

1). An expression which is not part of any other expression has bias 0. The same applies to arguments to user-defined routines and to array index expressions.

2). The bias B of an expression which is an argument of a predefined operator with bias B0 is determined by one of the four rules below:

```
    SAME:   B = B0
OPPOSITE:   B = -B0
      -1:   B = -1
       0:   B = 0
```

Which rule is used is specific to the operator and to the position of the argument (first argument, second argument etc.).

In general, how an argument to an operator "inherits" its bias from that of the operator has been defined according to the principles

1. If an argument must evaluate to an integer value according to the rules of the language, it is computed with zero bias (rule "0").

2. Otherwise, consider the partial derivatives of the corresponding ideal operator in the region

$$R = \{ (x_1,\ldots,x_n) \mid x_1 \geqq 0,\ldots,x_n \geqq 0 \}$$

(n is the arity of the ideal operator). If the k'th partial derivative is $\geqq 0$ throughout R (i.e. if the result of applying the operator is a non-decreasing function of the k'th argument, considering only positive arguments), rule "SAME" is used for the k'th argument. If the partial derivative is $\leqq 0$ throughout R, rule "OPPOSITE" is used.

3. Arguments which serve as error bounds are computed with bias -1 (rule "-1").

This gives rise to the following table:

Operator	1.argument	2.argument	3.argument
+ (dyadic)	SAME	SAME	
- (dyadic)	SAME	OPPOSITE	
*	SAME	SAME	
/	SAME	OPPOSITE	
+ (monadic)	SAME		
- (monadic)	OPPOSITE		
↑	SAME	0	
DIV , MOD	0	0	
< , <=	OPPOSITE	SAME	
> , >=	SAME	OPPOSITE	
= , <>	SAME	SAME	
NOT	OPPOSITE		
AND , OR	SAME	SAME	
sum , prod	SAME	SAME	-1
dif , quot	SAME	OPPOSITE	-1
lt , le	OPPOSITE	SAME	-1
gt , ge	SAME	OPPOSITE	-1
abs	SAME		
ent	SAME		
e	SAME	0	
expo	SAME		
copy	SAME	-1	

Table 5. Bias inheritance.

2.7 Miscellaneous.

PAL contains no type declarations. A variable declaration looks like e.g.

```
VAR x,y,a[1..4,1..4];
```

Here x and y are simple variables and 'a' a 4x4 matrix.

Arrays are dynamic in the Algol 60 sense, i.e. their bounds are computed on entry to the routine in which they are declared.

All simple variables and all elements of array variables are initialized to the value 0. This in particular applies to the special function identifier variables used (as in Pascal) to return values from functions.

PAL supports two ways of passing parameters to routines. One is intended to be reminiscent of call-by-value but could be called "call-by-constant": No copy of the actual parameter is created. Instead, the value of the actual parameter is accessible in the procedure body through the formal parameter. However, no assignment may be made to the formal parameter.

The other parameter passing mechanism is call-by-reference, specified as in Pascal by prefixing the formal parameter name by the key word 'VAR' in the routine heading.

3. Examples.

The square root function of section 1 can be recast in PAL as follows:

```
01  FUNCTION sqrt(x;eps);
02
03    CONST frac=0.1;
04    VAR y,old_y,d,c;
05
06  BEGIN
07    y:=initial_value(x);
08    c:=<1/(2*y);
09    d:=initial_deviation(x);
10    REPEAT
11      old_y:=y;
12      d:=<c*d*d;
13      y:=y+quot(x-y*y,2*y,frac*d);
14    UNTIL abs(y-old_y)<=eps;
15    sqrt:=y;
16  END;
```

Algorithm 2.

Remarks:

Line 01: The function name has been changed to 'sqrt' in accordance with the basic philosophy of PAL that "everything is multiple-precision".

Line 04: Variables d and c have been added to keep track of the deviation $|y_n - \sqrt{x}|$ of the computed iterands from $\sqrt{x}$, and to hold the constant $(2y_0)^{-1}$, respectively.

Line 08: Both operations are APC operations: there is no need to compute c accurately.

Line 09: d is set equal to some estimate of the deviation of initial_value(x) from $\sqrt{x}$.

Line 12: At the start of the loop, d is always approximately equal to the deviation of the current iterand from $\sqrt{x}$. For the ideal iteration we have

$$|y_{n+1} - \sqrt{x}| = \frac{1}{2y_n} |y_n - \sqrt{x}|^2 \quad .$$

Thus, after execution of line 12, d is approximately equal to the deviation of the next iterand from $\sqrt{x}$. Note also that if the bias operator '<' had been omitted, the multiplications would have been exact, leading to an exponential growth of the number of digits in d.

Line 13: Because of the quadratic convergence, we need all digits of y*y (PAL has no squaring operator) and

2*y. Thus the division is the only operation giving rise to a computational error. The permitted error can be seen to be a small fraction of the estimated deviation of the new iterand from $\sqrt{x}$. This is a reasonable choice which avoids useless work (a too accurate division), yet preserves the convergence properties.

The full power of the bias facility was not exploited in the above example; the bias operators could have been "reversed" ('<' replaced by '>') with no appreciable effect. This is because no attempt was made to prove that the algorithm computes $\sqrt{x}$ to within an error of *eps* (indeed, the algorithm, although usable in practice, is not provably correct). The following trivial example of an algorithm accompanied by a correctness proof assumes the availability of unrestricted algorithms for sin(x) and π . Specifically, the functions 'sin' and 'pi' must satisfy

$$V(\sin(x,f)) = \sin(V(x)) + V\hat{}(f) \quad \text{and}$$

$$V(\mathrm{pi}(f)) = \pi + V\hat{}(f) \quad ,$$

and the problem is to construct an unrestricted algorithm for cos(x). Using the relation

$$\cos(x) = \sin(\pi/2-x) \quad ,$$

a PAL solution might be

```
FUNCTION cos(x;f);
BEGIN
  cos:=sin(dif(0.5*pi(<0.2*f),x,0.1*f),<0.8*f);
END;
```

Algorithm 3.

and the correctness proof goes as follows:

$$V(\cos(x,f)) = \sin(V(\mathrm{dif}(0.5*\mathrm{pi}(<0.2*f),x,0.1*f)) + 0.8V\hat{}(f) =$$

$$\sin(V(0.5*\mathrm{pi}(<0.2*f)) - V(x) + 0.1V\hat{}(f)) + 0.8V\hat{}(f) =$$

$$\sin(0.5*(\pi+0.2V\hat{}(f)) - V(x) + 0.1V\hat{}(f)) + 0.8V\hat{}(f) =$$

$$\sin(\pi/2 - V(x) + 0.2V\hat{}(f)) + 0.8V\hat{}(f) = \cos(V(x)) + V\hat{}(f) \quad .$$

If the bias of one of the three error bounds were changed from -1 to 1, the proof would collapse. If a bias were

changed to 0, the proof would still work; however, the algorithm would waste much time doing multiplications by f when f contains many digits.

4. Discussion.

The three shortcomings of traditional programming languages have been entirely remedied:

- A PAL program is a complete description which can be fed directly into a computer.

- An error bound is specified (explicitly or implicitly) with each primitive operation. The reader should, however, be aware that this is not the ideal state of affairs: It would be a considerable improvement if the system could deduce this detailed information from a description at the statement level or perhaps even above the statement level of a global required accuracy. However, the selection of appropriate error bounds often requires a good deal of mathematical insight into the algorithm at hand, and it is difficult to see how this step might be automated.

- The discrete measure of "computer words" has been replaced by the continuum of real absolute error bounds.

4.1 Basic principles.

PAL was designed according to the principle that the programmer of multiple-precision algorithms should be aware that precision costs time and should always understand his algorithm well enough to know how much accuracy is needed in its various parts. In addition, it should be possible to carry out inexpensive operations (giving, of course, less accurate results) even on numbers of a high stored precision, rather than have accurate numbers blindly force all operations to be of corresponding precision. This is the principle of transparent precision.

It is clear that a system to perform multiple-precision computations must require the user to state at some point how much accuracy he actually needs. Many multiple-precision

packages are simply committed at compile-time to a particular size of multiple-precision variables and to operations of corresponding accuracy. However, this is a very crude mechanism. Languages like Cobol, ADA and PL/I allow a different precision to be associated with each variable. But this cannot cope with the situation in which the same statement is executed repeatedly with increasing precision, such as in the example square root routine. The only way to attain the necessary flexibility is to let precision be specified (and changed) dynamically. This can be done in several ways. One is to have a procedure 'set_precision', a call

```
set_precision(100)
```

meaning: "Use a precision corresponding to 100 decimal digits (e.g.) until the next call of 'set_precision'". The different strategy adopted in PAL is to have a parameter specifying precision associated with each call of a predefined operator. To have this parameter be just one more operand of the same type as the other operands is convenient to the programmer and lends a pleasing uniformity to the language. Since there seems to be no nice syntactic way to accomodate a third operand to an operator like '+', usual function syntax is used (e.g. "sum(x,y,f)"). This has the unfortunate effect of rendering some expressions very unreadable. However, this applies only to F operators, and as the examples show, these occur relatively unfrequently.

The operand specifying precision invariably states the maximum allowable absolute error. One might instead have employed a bound on the relative error, a logarithm of the latter (corresponding to the number of bits or decimal digits), or something quite different. The absolute error bound was chosen because of its simplicity and because the relative error creates problems around zero.

4.2 The predefined operators.

Only the most basic mathematical operators have been included in the language itself. An implementation may provide elementary mathematical functions in the form of a standard library written in PAL. The examples above have shown appropriate conventions for functions like sin(x), π etc.

Among the predefined operators most are completely traditional (disregarding their multiple-precision nature and a possible error bound parameter). 'expo' and 'e' are indispensable, e.g. for the purpose of scaling. They may be

thought of as approximate logarithm and exponential functions, respectively. Alternatively, they may be viewed as orderly ways to inspect and to change, respectively, the otherwise inaccessible internal exponent of PAL reals.

4.3 APC operators.

The usefulness of the APC operators has already been demonstrated. As a further example, the reader may try to reformulate algorithm 2 without the use of APC operators. APC operators are clearly most useful in the computation of error bounds, as opposed to "data".

The rounding offered by the APC operators greatly facilitates proofs of algorithm correctness. The rounding can generally be directed so that the errors associated with the APC operators affect (marginally) speed, but not correctness.

The accuracy of the APC operators is easily seen to closely mirror the accuracy one would expect from the real arithmetic of a computer which uses

$$-\log_2 \Delta$$

bits for the mantissa (here Δ means the APC delta). This makes the APC axioms easy to remember and to use. For maximum efficiency on a given machine, Δ can often be chosen close to 2^{-n}, where n is the number of bits per word of the machine, since APC operators, given this choice, need only manipulate one or two words of their multiple-word arguments. This is the reason why each implementation has been allowed to choose its own Δ. On the other hand, this causes a very unpleasant lack of portability: A PAL program making assumptions about a particular APC delta can only be moved safely to implementations with smaller values of Δ. Since the actual size of Δ is of very little consequence to the PAL programmer as long as it is not too large, it might be preferable instead to have the language definition stipulate one particular (fairly large) Δ, such as $\Delta = 10^{-3}$, the idea being that any implementation is likely to be able to provide low-cost APC operators for a Δ of this magnitude.

4.4 Bias inheritance.

The implicit bias inheritance scheme of PAL contributes considerably to the ease with which APC operators may be

combined as well as to the readability of the resulting expressions.

The four principles of section 2.5 according to which the bias inheritance has been defined are quite natural, except for the proviso concerning the region R. The bias of an expression may be taken as an indication of the programmer's intention as to how rounding should be performed during evaluation of the expression. Arguments which contribute positively to the final result (such as the first argument of a subtraction) should be evaluated with the same kind of rounding, whereas the opposite rounding should be used for arguments which contribute negatively to the final result (such as the second argument of a subtraction). Expressions which must evaluate to integer values are given bias 0 because expressions containing APC operators cannot in general be guaranteed to yield integer values, and because the evaluation of integer-valued expressions normally is fast, even if exact operators are used. Finally, arguments which serve as error bounds are given bias -1 because it is desired that the computed error bound is at most as great as the ideal value.

The restriction of consideration to the region R of positive argument values is necessary in order to keep the bias a static quantity (consider a multiplication with arguments of varying signs). The reason why this restriction is sensible is that most expressions with a non-zero bias have to do with the computation of error bounds and have sub-expressions which are known to be positive (compare the examples). In any case, the default bias inheritance rules are only meant as a help to the programmer and can be overridden where necessary.

4.5 Types.

The elimination of types from PAL may be seen by some as a retrograde step. On the other hand, most numerical algorithms seem to make little use of other types than integers, reals (and arrays of these) and, to a lesser extent, booleans. Traditionally, integers and reals are treated as disjoint types. In PAL, integers are simply special values which may be assumed by any PAL real, just as the mathematical set of integers is a subset of the set of real numbers. Of course, an implementation may, if it so wishes, try to spot variables used as integers or booleans and other candidates for optimization.

4.6 Parameter passing mechanisms.

Call-by-value was replaced by "call-by-constant" in order to preserve the principle of transparent precision at the level of user-defined functions. Call-by-value requires that a copy of the actual parameter be created (to full accuracy), regardless of how accurately the value of the parameter is actually needed in the routine. In contrast, the run-time implementation of "call-by-constant" can be almost exactly as for call-by-reference.

4.7 Implementation.

The definition of PAL says that operations of arbitrary accuracy can be carried out on arbitrary PAL reals. It is clear that no implementation can fully comply with this requirement. But a good implementation should try to impose few other restrictions than those dictated by the finite (total) amount of storage available to the system. This means that it must be possible to dynamically vary the amount of storage allocated to each variable. In any case, no computation may be allowed to proceed after a failure by a predefined operator to satisfy one of its defining axioms.

The range of available operators probably forces the implementation to represent multiple-precision numbers in some sort of positional system if it is not to be excessively slow; a representation of this kind has been tacitly assumed a number of times above. This dependency on a particular representation is regrettable; on the other hand, the vast majority of existing multiple-precision systems employs representations based on positional systems.

The set of numbers having finite representations in a given positional system is not closed with respect to division. This is the reason why PAL contains no exact division operator. There are other similar restrictions which have not been mentioned. For instance, the second operand to the exact exponentiation operator may not be negative.

The operators 'expo' and 'e' are intended to be very fast. Unfortunately, this can be achieved only if the base of the internally used positional system is a power of 10 (10 having been chosen as the most convenient base for humans). In practical terms, this is a much more severe restriction on the implementation since internal bases more often are powers of 2. The language definition might have circumvented the problem by permitting each implementation

to provide its own base to replace the constant 10 in the definition of 'expo' and 'e'; this was rejected because of the adverse effects it would have on portability.

5. An extensive example.

For a more substantial example of a PAL routine, consider the following more elaborate unrestricted square root function which is provably correct and considerably more efficient than algorithm 2. The workings of the algorithm will not be explained here.

```
FUNCTION sqrt(x;f);

(* computes the square root of x
   to within an absolute error bound of f *)

  CONST c1=0.1245;
        c2=1.17;
        c3=4;
        c_4=4;
        c_5=-1;
        c6=0.01;
        c7=0.28;

  VAR g,i,k,n,u,v,y;

  FUNCTION b;
    VAR m;
  BEGIN
    IF x<0 OR f<=0
    THEN
      BEGIN
        IF x<0
        THEN writeln(' %%% sqrt(x,f) called with x<0')
        ELSE writeln(' %%% sqrt(x,f) called with f<=0');
        terminate;
      END;
    m:=expo(x) DIV 2; (* for the purpose of scaling *)
    y:=<e(c1,-m)*(x+e(10,m+m));
    (* linear initial approximation *)
    g:=<c2*f/y;
    k:=expo(g);
    b:=c3*expo(k)+c_4; (* b+1 is an upper bound on the
                          number of iterations *)
  END;
```

```
    VAR a[1..b];  (* used as a stack *)

BEGIN (* sqrt *)
  IF x<>0
  THEN
    BEGIN
      (* n=0 *)
      (* compute required accuracies of iterations
         in reverse order - push them on 'a' *)
      WHILE k<c_5
      DO BEGIN
           n:=n+1;
           k:=k DIV 2;  (* note: rounding away from 0
                           since k<0 *)
           a[n]:=k;
         END;
      u:=<c6*y*y;
      v:=<c7*y;
      FOR i:=n DOWNTO 1
      DO y:=y+quot(dif(x,y*y,e(u,a[i])),y+y,e(v,a[i]));
      (* pop accuracy from 'a' - compute next iterand *)
      sqrt:=y+quot(dif(x,y*y,u*g),y+y,v*g);
      (* last iteration *)
    END;
END;
```

Algorithm 4.

Summary.

A new programming language PAL to describe multiple-precision computations has been introduced and has been shown to possess a number of attractive characteristics.

All variables in PAL are of the same type, and the programmer need not worry about storage allocation. PAL contains F operators, exact to within a specified error bound, exact operators, and APC operators which are similar to single-precision real operations but use directed rounding. A set of default rules helps the programmer specify how rounding should be performed in composite expressions.

Acknowledgements. I am indebted to several colleagues at the University of Århus, in particular Ole Østerby.

References.

[1]: Richard P. Brent: "Multiple-Precision Zero-Finding Methods and the Complexity of Elementary Function Evaluation". In: "Analytic Computational Complexity" (editor: J.F.Traub), pp. 151-176. Academic Press, New York, 1976.

[2]: Torben Hagerup: "Calculation of Elementary Mathematical Functions to an Arbitrary Precision". M.S. thesis. Internal report DAIMI IR-48, University of Arhus, May 1983.

[3]: Torben Hagerup: "PAL Manual". Internal report DAIMI MD-48, University of Arhus, August 1983.

[4]: F.W.J.Olver: "Unrestricted Algorithms for Generating Elementary Functions". Computing Supplementum 2: "Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis)" (edited by G. Alefeld and R.D.Grigorieff), pp. 131-140. Springer-Verlag, Wien, 1980.