Some Applications of Continuations

L. ALLISON
Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia

Continuations are used in denotational semantics to describe control commands such as jumps. Here it is shown how they can be used as a programming technique to simulate backtracking and co-routines.

Received December 1986

1. INTRODUCTION

Continuations are used in denotational semantics[7,9] to describe the semantics of control mechanisms and of control commands such as jumps. Continuations are an example of a sophisticated use of high order functions. Strachey and Wadsworth[9] give their origins in the tail functions of Mazurkiewicz[5] and of Morris. Here, continuations are used as a programming technique to simulate non-determinism and co-routines either in a functional language or when using a functional style in an imperative language. It is hoped that this will further the use of this powerful tool from the functional programming armoury.

Functional composition, o, is the usual way of combining two functions:

$$f{:}B\rightarrow C,\quad g{:}A\rightarrow B$$

$$f\circ g x = f(gx).$$

Two composed functions pass an intermediate value between themselves. Continuations give another way to combine functions. There is a function g' related to g:

$$g'{:}(B\rightarrow C)\rightarrow A\rightarrow C$$

$$g'fx = f(gx)$$

Note that $f\circ g = g'f$. f is a continuation to g'. In this case, g' passes a value (gx) and control to f. There is no o operator involved. g' has the extra freedom to manipulate the flow of control and it will be exploited in the following sections.

2. NON-DETERMINISM

Non-determinism appears where that may be zero, one or many solutions to a problem and where a search is necessary to discover which will succeed. This section illustrates the implementation of non-determinism by continuation functions via an example from parsing. Non-determinism is a central feature of logic programming languages such as Prolog.[3] The technique of implementing non-determinism in a functional language is present in the various denotational semantics of Prolog[4,8] and in implementations of Prolog in functional programming[2] but it deserves wider exposure. The use of continuations is implicit in the organization of the trail stack in a conventional implementation of Prolog.

The following grammar is non-deterministic:

$$\langle S\rangle ::= \langle aORaa\rangle\langle aORaa\rangle$$

$$\langle aORaa\rangle ::= a\,|\,aa.$$

The string 'aaa' has two parses – a(aa) or (aa)a. Such a grammar cannot usually be parsed by recursive descent because of the non-determinism. However, recursive descent plus continuations can do the job.

The following example indicates how this grammar can be parsed. It is written in a simple functional language. The answers returned by the parser are just indications of success but the parse trees could easily be returned. First some general operators are defined:

```
Types or domains:
  Ans = {Success}*
  input: Text = char*
  cont: Pcont = Text→Ans
  Parser = Pcont→Text→Ans = Pcont→Pcont

fin input =
  if null input then [Success]
  else nil
fin: Pcont

letter ch cont input =
  if null input then nil
  else if ch = hd input then
    cont tl input
  else nil
letter: char→Parser

seq p1 p2 cont = p1 (p2 cont)
seq: Parser→Parser→Parser

either p1 p2 cont input =
  append (p1 cont input)
         (p2 cont input)
either: Parser→Parser→Parser
```

Using these operators, the particular grammar can be programmed:

```
a = letter 'a'
aORaa = either a (seq a a)
S = seq aORaa aORaa
a, aORaa, S: Parser
```

The expression 'S fin string', will indicate how many parses the string has.

Each parser has a parser continuation, a Pcont, which follows it. The function 'seq' forms the sequential composition of two parsers, as indicated by concatenation in BNF. The definition of seq exactly follows that of sequential composition (;) in the standard semantics of programming languages. To parse 'p_1' and then 'p_2' and finally do 'cont', call p_1 with a continuation which is (p_2 cont). The function 'either' performs alternation, '|'. It simulates non-determinism by trying both alternatives with the following continuation and appending results.

If only one solution is required, fin should be modified to raise an exception which should be caught by S.

2.1. Pascal

The central idea in the parser of the previous section is that of parser continuations, Pcont. These can be programmed in Pascal. The only slight difficulty is the lack of curried functions (or procedures). This means 'seq' and 'either' cannot be used in partially parameterized form which renders them rather pointless; suitable in-line code is more appropriate.

```
program Continuations(input, output);
var l:array[1..80]of char;
                              {the input string}
    len:integer;                    {its length}

procedure fin(m:integer);
begin if m=len+1 then writeln('Success')
end;

procedure aORaa(
  procedure cont(p:integer);
  n:integer);
begin if l[n]='a' then
      begin cont(n+1);                      {a }
            if l[n+1]='a' then
               cont(n+2)                    {aa}
      end
end;
procedure sentence(
      procedure cont(p:integer);
      n:integer);
  procedure aORaaCont(n:integer);
  begin aORaa(cont, n) end;
begin aORaa (aORaaCont, n) end;

begin writeln('type a string');
  while not eof do
  begin len:=0;
      while not eoln do
      begin len:=len+1; read(l[len])
      end;
      readln; writeln;
      sentence(fin, 1);
      writeln('finished')
  end
end.
```

In this version the input is held in a global buffer 'l'. Because Pascal is imperative it is possible, although not essential, for the final continuation 'fin' to act as a side effect.

If it is only necessary to accept the first parse out of many, procedure fin should be modified to do a non-local goto out of the parser.

2.2 Parsers

In parsing terms, the original grammar requires a slow-back top-down parser. The string 'aaa' can be parsed as a(aa) or (aa)a. The first aORAa can succeed by parsing 'a' but its activation cannot be discarded because it may be required to succeed again as 'aa'. When one of the parsers is run, the first aORaa in a sentence calls the second one which is incorporated in the first's continuation. The first one is still active and can succeed again in another way.

3. CO-ROUTINES

A set of co-routines is a collection of routines which pass control amongst themselves. A co-routine is *resumed* from the point at which it was last suspended; apart from its initial start it resumed just after the point that it last resumed some other co-routine. Simula and Modula provide co-routines. In this section the example of merging two search trees will be used to illustrate how continuations can simulate co-routines.

A search tree is a binary tree in which the elements in the left subtree are less than the element in the root which is less than the elements in the right subtree. This property also applies to all subtrees. The problem is to merge the elements of two search trees so as to print one ascending list of their elements. The natural solution to this problem uses two co-routines. Each co-routine traverses one tree in infix order. Co-routine A resumes co-routine B when the element that A is examining exceeds the element that B is examining.

There are many other solutions such as

```
λ tree₁, tree₂. listmerge (flatten tree₁)
                          (flatten tree₂)
```

but these all produce temporary structures or visit some nodes more than once. The tree merge algorithm given here simulates the co-routine solution.

3.1. Tree flattening

The tree merge algorithm is based on the following unusual way of flattening a single tree.

```
Type:
  cont: Cont=Void→List of element
flatten t=
  let fin ( )=nil,
      fl t cont=
        if null t then cont( )
        else
        let contright ( )=
              (elt t).(fl (right t) cont)
        in fl (left t) contright
  in fl t fin
flatten: Tree of element→List of element
fin, contright: Cont
fl: Tree of element→Cont→List of element
```

The continuation parameter 'cont' of 'fl' is an accumulating parameter. It is a function which will flatten the rest of the tree once fl has flattened the left subtree. Note that '.' is the infix list constructor. Flatten can easily be programmed in Pascal.

3.2. Tree merging

The tree-flattening function of the previous section can be used as the basis of a tree-merging function. To merge two trees first find their smallest (left most) values, then traverse one tree until an element larger than a 'switch' value is met. Then an 'alternative' function must be invoked. The alternative will traverse the other tree until a switch back to the first tree is necessary. The alternative is the co-routine or co-function of the current traversal function.

```
Type:
  Cont = element→Cont→List of element

merge tree₁ tree₂ =


let rec
  traverse tree cont switch alternative =
    if null tree then
       cont switch alternative
    else traverse (left tree)
         (travright tree cont)
         switch alternative,

  travright tree cont switch alternative =
    if null tree then
       cont switch alternative
    else if elt tree≤switch then
      (elt tree)
      .(traverse (right tree)
         cont switch alternative)
    else alternative (elt tree)
        (travright tree cont),        —†

  m tree₁ cont₁ tree₂ cont₂ =
    if not null (left tree₁) then
      m (left tree₁)
        (travright tree₁ cont₁)
        tree₂ cont₂
    else if not null (left tree₂) then
      m tree₂ cont₂ tree₁ cont₁
    else if elt tree₁≤elt tree₂ then
      travright tree₁ cont₁ (elt tree₂)
        (travright tree₂ cont₂)
    else
      travright tree₂ cont₂ (elt tree₁)
        (travright tree₁ cont₁),
  fin switch cont = cont maxint halt,
  halt switch cont = nil
in
  if null tree₁ then
    traverse tree₂ halt maxint halt
  else if null tree₂ then
    traverse tree₁ halt maxint halt
  else m tree₁ fin tree₂ fin
```

```
fin, halt: Cont
traverse, travright:
   Tree of element→Cont→Cont
m: Tree of element→Cont→Tree of
      element→Cont→List of element
merge: Tree of element→Tree of
         element→List of element
```

A continuation simulates a co-routine. Its element parameter is a parameter of the resumption. The Cont parameter is the other co-routine, itself to be resumed at a later date. When the alternative is resumed (†) it is given a switch parameter and the 'current' co-routine. Note that travright is not a continuation but

```
travright tc: Cont
  if t:Tree of element, c:Cont
```

If one of the trees is empty, traverse the other tree. Otherwise 'm' finds the least element of tree$_1$ (and later tree$_2$) while building a continuation to traverse tree$_1$. When the continuation for each tree is built, the appropriate one is started and the two of them swap control as necessary.

3.3. Pascal and Algol

The tree-merge function cannot be programmed in Pascal because this language does not allow recursive function types such as 'Cont'. However it can be programmed in Algol-68 which does allow such types.

McVitie and Wilson give an Algol-60 program for the stable marriage problem (McVitie and Wilson,[6] algorithm 2 and §3.2). It uses recursive procedure calls in a way that mimics continuations. Coincidentally, it appeared at about the same time as Mazurkiewicz's paper.[5] The stable marriage problem has a natural solution by co-routines.[1]

4. CONCLUSION

The use of continuations allow a function to govern the flow of control into and out of what would normally be considered a subsequent computation. This allows unusual control regimes such as non-determinism and co-routines to be simulated in functional programming. In particular, non-deterministic grammars can be parsed by simple recursive descent plus continuations and functions can be made to co-operate as co-routines.

REFERENCES

1. L. Allison, Stable marriages by co-routines. *Information Processing Letters* 16, pp. 61–65, Feb. 1983.
2. M. Carlsson, On implementing Prolog in functional programming. *1984 International Symposium on Logic Programming*, pp. 154–159.
3. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. Springer Verlag 1981.
4. N. D. Jones and A. Mycroft, Denotational semantics of Prolog. 1984 *International Symposium on Logic Programming*, pp. 281–288.
5. A. W. Mazurkiewicz, Proving algorithms by tail functions. *Information and Control* **18**, 220–226 (1971).
6. D. G. McVitie and L. B. Wilson, The stable marriage problem. *CACM*, 14, no. 7, pp. 486–490 (July 1971); and algorithm 411, *CACM* vol. 14, no. 7, pp. 491–492 (July 1971).
7. R. Milne and C. Strachey, *A Theory of Programming Language Semantics*, 2 vols. Chapman Hall (1976).
8. T. Nicholson and N. Foo, A denotational semantics for Prolog. Basser Dept. Computer Science, University of Sydney (1985).
9. C. Strachey and C. P. Wadsworth, Continuations, a mathematical semantics for handling full jumps. PRG-11 Oxford University (1974).

Test Procedures for Measurement of Floating-Point Characteristics of Computing Environments

M. RAZAZ AND J. L. SCHONFELDER†

Department of Electronic and Electrical Engineering, The University of Birmingham, PO Box 363, Birmingham B15 2TT
† *University of Liverpool, PO Box 147, Liverpool L69 3BX*

A number of test procedures are presented for measuring the floating-point characteristics of a processor in a given computing environment. By using these procedures, accurate values can be assigned to the number representation and precision-dependent parameters such as the normalisation base, number of digits in the mantissa, nominal decimal precision, minimum representable number and so on. The procedures also determine the statistical properties of literal and input conversions and the basic arithmetic operations on the processor. Typical experimental results for the DEC, IBM, CDC and ICL computers are presented and discussed

Received December 1986, revised April 1987

1. INTRODUCTION

In a computing environment, a set of machine-dependent parameters can be identified which indicate the properties and limits of the floating-point arithmetic on the processor. A possible set includes such parameters like the normalisation base, the overflow and underflow thresholds, the number of digits in the mantissa, the relative precision and so on.[1] Assignment of proper values to such parameters is vital for accurate floating-point computation and for producing transportable numerical software.[2-4] By using normal manufacturers' documentation it should be possible, in principle, to find values for these parameters.[13-15] However, in practice this may not be true for all cases. To determine the radix employed and the number of digits retained in the mantissa is normally a straightforward task. It would be more difficult, however, to find in all cases details of arithmetic round or chop or the number of guard digits retained in intermediate calculation, and in particular how accurately external decimal values are converted to internal storage forms. The latter process is particularly important in the production of portable numerical software involving a large number of literal constants.[3,4]

This paper presents a number of test procedures which measure accurately such parameters and determine the statistical behaviour of the basic arithmetic operations and of conversion from external decimal value to internal storage form. These procedures were successfully implemented in a transportable Fortran software package,[5] which has been tested by one of the authors (MR) on some twenty major computers at different sites in the UK and abroad. The scope of this testing activity is described in Ref. 6. Although it has been possible to examine thoroughly the precision-related parameters, transportability is still not perfect as far as accurate measurement of the range thresholds is concerned. This is because (i) any attempt to test them directly would cause a threshold violation and (ii) various processors differ widely in their approach to handling floating-point exceptions. However, an indirect approach has been used to estimate their values. Software tools similar to those of Ref. 5 have also recently appeared in the literature.[16-18].

In Section 2 the details of the test procedures are described. Section 3 deals with typical test results obtained for a selection of processors, and finally conclusions are presented in Section 4.

2. TEST PROCEDURES

2.1 Representation test procedure

A test procedure is described in this section which can measure the parameters appropriate to stored floating-point quantities, i.e. the representation parameters. These are the radix or normalisation based α, the number of digits in the mantissa, d, and the minimum representable floating-point number, i.e. relpr. Values of d and relpr depend on the machine working precision. The procedure uses a modified algorithm first suggested in Ref. 7 and later enhanced in Ref. 8. A real number X is first constructed to satisfy $\alpha^d < X < \alpha^{d+1}$. It can be shown[7] that the radix $\alpha = X - Y$, where Y is constructed such that $X + Y \neq X$ and $X + Y/2 = X$. Having found α, the algorithm then determines whether the arithmetic rounds by testing if $(\alpha - 1) + X$ is different from the stored value of X. The algorithm thus detects rounding for the addition operation but not its form, nor whether the other arithmetic operations are also rounded. The value of d can then be evaluated from the smallest exponent of α^d such that

$$(((\alpha^d + 1.0) - \alpha^d) - 1.0) \neq 0.0.$$

A variant of the approach suggested in Ref. 8 is also used to force-store the results of each dyadic arithmetic operation for any subsequent comparison or operation.[6] Since the algorithm checks for rounding or otherwise, it is also used to infer relpr. The nominal decimal precision can then be calculated from $Nd = \{-\log_{10}(\text{relpr})\}$, where $\{A\}$ denotes the largest integer less than the floating-point number A.

2.2 Conversion test procedures

The conversion process in a computing environment involves representing a mathematically 'exact' real number by an internal machine value using a finite decimal floating-point approximation. Ideally, this process should be such as to ensure that the internal value is the nearest 'exact' value to the external number, as in Ref. 12. Although the ideal case may not be possible, a 'faithful' conversion can be obtained provided a sufficiently long decimal approximation is used.[9,6] In fact, a 'faithful' representation is the best that can be obtained in a correctly implemented conversion process. The number of decimal digits required for 'faithful' representation, Nc, can be obtained[6] from $Nc > 1+d\log_{10}\alpha$. It should be noted that choosing Nc to be greater than the minimum value will provide little or no improvement in conversion.[9,6] However, when 'faithful' conversion on a machine is not properly implemented, for example due to an artificial truncation of the decimal length, representation of literal or input constants in a program would become an important source of error and might complicate error analysis and/or software verification.[6] This section presents test procedures which enable us to detect any conversion errors and to determine statistically how well the conversion process is performed. The tests deal with literal conversion as well as floating-point numbers read by formatted input, that is, input conversion. The procedures for both cases are similar, and they are based on conversion and statistical error analysis of specially prepared sets of data. The latter consists of a large number of data samples, each containing a set of uniformly distributed random numbers which have been rounded to various decimal digits, starting at 5D and increasing to 36D. For each sample, a basic data set is then produced which contains an accurate reference standard, R, and six test values approximated from R to $Nd-1$, Nd, $Nd+1$, $Nd+2$, $Nd+3$ and $Nd+4$ decimal places, respectively.

As each test procedure is designed to work entirely with a single machine working precision, an approach had to be devised to produce R to extra accuracy. This was achieved by generating values of R with 40-decimal-digit accuracy using an Algol 68 multiple-precision package available at the computing centre of the University of Birmingham.[10] Once the required data structure is established, the procedure then compares the test values in a single data sample with the corresponding reference standard R in order to evaluate individual relative conversion errors. This process is repeated for the total number of data samples to provide, for a given working precision, the conversion-error statistics, namely the mean, rms (root mean-squared) and maximum errors.[6]

These conversion test procedures have been successfully implemented in a transportable software which is capable of testing any machine with nominal decimal accuracies ranging from 6D to 32D, that is, single, double and extended precisions.

2.3 Arithmetic test procedure

This section describes a test procedure which can measure the statistical properties of the basic arithmetic operations, that is, addition, subtraction, multiplication and division on a processor. Assuming that the operands themselves are exact, the test uses a higher working precision as a reference standard for testing a lower working precision. A pair of random floating-point numbers, Y and Z, are generated which are then assigned to variables of higher precision DY and DZ. This is followed by performing

$$B = Y\theta Z$$

$$DB = DY\theta DZ$$

for each operator $\theta = +, -, *$ and $/$. The relative error in B is then evaluated in higher precision as

$$\delta = (B-DB)/B$$

This process is repeated for a sufficiently large number of trials to take account of any statistical fluctuations. For each operation the mean, rms and maximum error statistics are then calculated and displayed. This test procedure, however, cannot determine the accuracy of the arithmetic operation for the highest working precision as a reference standard for testing a lower working precision.

3. EXPERIMENTAL RESULTS AND DISCUSSIONS

This section presents typical experimental test results for DEC, IBM, CDC and ICL computers, selected from a comprehensive set of results which the author has obtained on various processors. A detailed survey of these results will be the subject of a separate paper.[11]. The objective here is to demonstrate typical capabilities of the test suite, and to discuss and highlight some of the properties and peculiarities of the relevant floating-point characteristics which have been observed. The section is divided into four parts. Section 3.1 deals with the experimental results obtained from the number-representation tests. Sections 3.2 and 3.3 discuss the results associated with the literal and conversion tests. Finally the results of the arithmetic tests are presented in Section 3.3.

3.1 Representation test results

This section deals with some selected results obtained from representation tests. Table 1(*a*) shows the test results for the CDC 7600 machine with FTN 4.7 compiler. Entries in columns 2 and 3 are associated with single-precision (SP) and double-precision (DP) values. The computed values of Nd for single and double precisions are 14D and 28D, respectively. All the results obtained from this tests are as would be expected from the CDC documentations.

Table 1. Results of the representation test on three computers

(*a*) CDC 7600	SP	DP
α	2	2
d	48	96
Addition option	CHOP/ROUND	CHOP
Implied relpr	7.11E-15/3.55E-15	2.52D-29

(*b*) ICL 2980	SP	DP	QP
α	16	16	16
d (equivalent binary digits)	6 (24)	14 (56)	28 (112)
Addition option	CHOP	CHOP	CHOP
Implied relpr	9.54E-7	2.22D-16	3.08D-33

(*c*) DEC 20	SP	DP
α	2	2
d	27	62
Addition option	ROUND	ROUND
Implied relpr	7.45E-9	2.17D-19

Table 2. Results of conversion tests on three computers. ( ) denotes input conversion

No. of decimals	Error Mean	rms	Maximum	Bound
(*a*) CDC 7600				
13	5.3E-15	1.14E-13	4.56E-13	5.04E-13
14	−9.5E-16	1.14E-14	3.70E-14	5.36E-14
15	2.7E-17	1.96E-15	5.61E-15	8.55E-15
16	4.2E-17	1.52E-15	3.55E-15	4.05E-15
17	3.7E-17	1.52E-15	3.51E-15	3.60E-15
18	3.7E-17	1.52E-15	3.51E-15	3.56E-15

No. of decimals	Error Mean	rms	Maximum	Bound
(*b*) ICL 2980				
14	−(7.5) 75E-16	(1.2) 1.2E-14	(3.7) 3.7E-14	5.0E-14
15	(1.1) 1.1E-16	(1.2) 1.2E-15	(4.9) 4.9E-15	5.1E-15
16	(3.5) 3.5E-18	(1.2) 1.2E-16	(4.3) 4.3E-16	6.1E-16
17	−(2.8) 2.8E-18	(3.2) 3.2E-17	(1.1) 1.1E-16	1.6E-16
18	−(0.27) 1.8E-17	(2.9) 3.5E-17	(0.96) 1.1E-16	1.2E-16
19	−(0.50) 1.8E-17	(2.9) 3.6E-17	(0.96) 1.1E-16	1.1E-16

No. of decimals	Error Mean	rms	Maximum
(*c*) IBM 370			
14	−(7.47) 10.47E-16	(1.16) 1.12E-14	(3.75) 3.75E-14
15	(1.07) 0.915E-16	(1.21) 1.19E-15	(4.87) 4.87E-15
16	(3.52) 7.31E-18	(1.18) 1.13E-16	(4.32) 4.32E-16
17	−(2.83) 2.72E-18	(3.16) 3.14E-17	(1.11) 1.11E-16
18	−(2.75) 2.97E-18	(2.86) 2.81E-17	(9.58) 8.65E-17
19	−(4.96) 4.48E-18	(2.89) 2.83E-17	(9.58) 8.65E-17

Table 1(*b*) and 1(*c*) present the representation test results for the ICL 2980 machine (Fl.B.60 compiler) and DEC 20 machine (FTN 20 compiler). Note that the entries in the fourth column of Table 1(*b*) correspond to quadruple-precision (QP) values. The nominal precision for the two machines are obtained as $Nd = 6$, 15, 32 and $Nd = 8$, 18, respectively. Again, as before all the results are in agreement with the relevant machine documentations.

3.2 Conversion test results

This section discusses some selected results for literal and input conversion tests. Table 2(*a*) shows the results obtained for single-precision literal-conversion tests on the CDC 7600 machine. The columns show the mean, rms and maximum error statistics calculated for a range of decimal values of lengths 13D to 18D. The last column gives the theoretical relative error bound involved in the conversion process.[6] These figures are fairly typical of the behaviour found for a correctly functioning conversion process. As would be expected, the first two rows show errors to be dominated by the rounding in the underlength decimal representation. The third row indicates that the errors are predominant in the decimal round-off, but now some small effect of the decimal–binary conversion is also noticed. At the decimal length 16D, which corresponds to Nc, the round-off error is largely due to the finite length of the binary, and thus we have a faithful representation.

Increasing the decimal length to 17D and 18D has minimal further effects, as would be expected. Although the conversion process appears to produce results with little bias in the errors, the compiler flags any constant of length 16D or more as overlength and gives a warning message. This seems to be unnecessary as the required length for a faithful representation Nc is clearly 16D. An entirely similar behaviour was noticed for the input conversion process, and the related test results were found to be identical to those given in Table 2(*a*). The results of double-precision literal- and input-conversion tests for the ICL 2980 machines are presented in Table 2(*b*). The figures in parentheses correspond to input-conversion tests. Here we have an example of a conversion process which is not behaving as expected.[6] The results for input conversion are identical to those for literal conversion down to the decimal length 17D, at which length the expected effects due to increased importance of the decimal–hex conversion are noticed. While the rms and maximum errors for input conversion continue decreasing down to the 'faithful' decimal length 18D, the corresponding figures for literal conversion at this length increase. This implies that there is a premature truncation in the decimal length for literal conversion and would suggest the use of $NC = 17D$ instead of 18D. The corresponding conversion tests carried out on IBM 370 (H compiler) performed very well and the input- and literal-conversion characteristics were essentially identical (Table 2(*c*)).

For almost all the machines tested, the results obtained for the single-precision conversion process were found to be consistent with 'faithful' representations. For a few machines, double-precision conversions, and in particular literal conversions, have not been implemented properly. In one or two cases actual errors in the conversions have been noticed.[6] One such case is that of the double-precision conversions on the ICL 1906A machine with hardware-extended precision and any XFIV/XFIH/XFEH compilers. This machine with $\alpha = 2$, $d = 74$ ought to have a nominal precision $Nd = 22$D and a faithful length $Nc = 24$D. Preliminary tests showed gross errors in double-precision literal conversion, with little or no improvement in accuracy for decimal lengths above 21D (see Table 3(*a*)).

Subsequent tests with different data samples, all possible and all negative literals, revealed that the trouble occurred for negative values only. Table 3(*b*) shows that the conversion of indirectly signed literals, represented as $-(+d.dd-\text{D-}dd)$, is performed correctly, while directly signed literals (see Table 3(*a*)) are converted as if the internal representation had only 68, instead of 74, bits in the mantissa.[6]

Table 3. Results of DP literal conversion test on the ICL 1906A

No. of decimals	Error Mean	rms	Maximum
(*a*) Normal sample values			
21	−2.41E-22	1.04E-21	1.25E-21
22	−6.7E-22	8.89E-22	1.25E-21
23	−6.12E-22	8.87E-22	1.25E-21
24	−6.12E-22	8.87E-22	1.25E-21
25	−6.12E-22	8.87E-22	1.25E-21
26	−6.12E-22	8.87E-22	1.25E-21
(*b*) Indirectly signed negative sample values			
21	1.73E-22	6.24E-22	7.73E-22
22	2.55E-22	6.77E-23	9.35E-23
23	2.55E-22	2.59E-23	3.02E-23
24	2.55E-22	2.59E-23	3.02E-23
25	2.55E-22	2.59E-23	3.02E-23
26	2.55E-22	2.59E-23	3.02E-23

3.3 Arithmetic test results

This section presents typical results of arithmetic tests on the DEC 20 and CDC 7600 machines. Table 4(*a*) gives the results of testing single-precision arithmetic operation on the DEC 20 machine. The total number of trials used for testing each operation is 10^4. These figures are typical of a machine on which the basic arithmetic operations (with round option) are performed correctly.[6] A fairly larger value for the mean error associated with addition operation indicates the effect of residual bias introduced by round-up on tie values, which is rather common for binary floating-point systems. All the other operations (except addition) show very little bias, with the mean values being much less than the rms values.

The results of single-precision arithmetic tests on the CDC 7600 machine with round and chop options are presented in Table 4(*b*). The figures in parentheses are for the round option. These results indicate a relatively poor arithmetic performance with a number of peculiarities.[6] One of its poor features is the lack of guard digits on subtraction operation. A close inspection of the CDC arithmetic shows that although a double-length intermediate results is actually used, the truncation to a single length occurs before post-normalisation, which has the effect of ignoring any guard digit. Another poor feature is that rounding reduces the maximum error by almost 75% rather than by 50% as would be expected. Close inspection of the pattern of rms and maximum errors in Table 4(*b*) and their comparison with those in Table 4(*a*) for the DEC 20 also reveals some inconsistencies in their statistical behaviour.[6]

4. CONCLUSIONS

A number of test procedures were presented which can be used to determine accurately the behaviour of the floating-point characteristics of a computing environment. These procedures, implemented in a transportable software tool, have been used for testing most of the major Fortran processors now in use. It can be said that the majority of these processors have produced test results consistent with good floating-point charac-

Table 4. Results of SP arithmetic test on two computers. ( ) Denotes the 'round option'

Operation	Error Mean	rms	Maximum
(*a*) DEC 20			
Addition	−1.64E-9	3.52E-9	7.44E-9
Subtraction	−6.37E-10	2.28E-9	7.45E-9
Multiplication	5.36E-11	3.17E-9	7.41E-9
Division	−8.75E-11	3.11E-9	7.40E-9
(*b*) CDC 7600 using ROUND and CHOP options			
Addition	(−0.335) 1.75E-15	(1.69) 2.34E-15	(4.86) 6.73E-15
Subtraction	(2.64) 3.39E-15	(8.63) 8.63E-14	(8.60) 8.60E-12
Multiplication	(0.716) 2.56E-15	(1.82) 3.02E-15	(5.21) 7.03E-15
Division	(−0.024) 2.54E-15	(1.67) 2.99E-15	(5.19) 7.08E-15

teristics. However, a few processors did not perform properly, by showing problems such as premature truncation of decimal values, errors in conversion or lack of guard digits on subtraction and so on.

Acknowledgements

The authors would like to acknowledge the considerable help given by many people at different computer sites in the UK and abroad, where facilities were provided to run these time-consuming tests.The financial support for this research project by the Science and Engineering Research Council of the UK is gratefully acknowledged. Finally, one of the authors (MR) would like to thank one of the referees for his constructive suggestions.

REFERENCES

1. W. S. Brown and S. I. Feldman, Environment parameters and basic functions for floating point computation. *ACM Trans. Math. Soft* **6**, 510–523 (1980).
2. P. Naur, Machine dependent programming in common languages. *BIT* **7**, 123–131 (1967).
3. X. Grees and X. Hartmanis (eds), *Portability of Numerical Software*. Lecture Notes in Computer Science. Springer Verlag, Heidelberg (1976).
4. J. L. Schonfelder, M. Razaz and M. S. Keech, High precision Chebyshev expansions for the modified Bessel functions. Technical Report, University of Birmingham, UK (1980).
5. M. Razaz and J. L. Schonfelder, A numerical transportable software package. University of Birmingham, UK (1982).
6. M. Razaz and J. L. Schonfelder, Measurement of numerical precision. Technical Report, University of Birmingham, UK (1983).
7. M. A. Malcolm, Algorithms to reveal properties of floating point arithmetic. *Comm. ACM Numeric. Maths.* **15**, 949–951 (1972).
8. W. M. Gentleman and S. B. Marovich, More on algorithms that reveal properties of floating point arithmetic units. *Comm. ACM Numeric. Maths.* **17**, 276–277 (1974).
9. D. A. Matula, Formalisation of floating point numeric base conversion. *IEEE Trans. Comp.* **19**, 681–692 (1970).
10. J. L. Schonfelder and J. T. Thomason, Arbitrary precision arithmetic in Algol 68. *Software Practice and Experience* **9**, 173–182 (1979).
11. M. Razaz, A survey of floating point characteristics of major computers. (Manuscript in preparation.)
12. ACRITH Documentation, IBM Program no. 5664–185, pp. 77 (1983).
13. *IBM System 370/Principles of Operation*. White Plains publication no. GA22-7000, chapter 9. IBM, White Plains (1983).
14. *CDC CYBER 170/180 Computer Systems, Hardware Reference Manual*. Publication no. 60469290, CDC, St Paul (1984).
15. *VAX Architecture Handbook*. Digital Equipment Corporation, Maynard, Mass., (1981).
16. N. Schryer, A test of a computer's floating-point arithmetic unit. AT & Bell Labs., Piscataway, N.J., Computer Science Technical Report 89 (February 1981).
17. B. A. Wichmann, FPV, a commercial software product available from the National Physical Laboratory and NAG Ltd (1986).
18. R. Kapinski, Paranoia: a floating point benchmark. *Byte*, 223–235 (February 1985).