THE AUSTRALIAN NATIONAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

TECHNICAL REPORT

AN IDEALIST'S VIEW OF SEMANTICS FOR INTEGER AND REAL TYPES

BY

RICHARD P. BRENT

TR-CS-81-14

An idealist's view of semantics for integer and real types

Richard P. Brent
Department of Computer Science
Australian National University
Box 4, Canberra, A.C.T., 2600.

TR-CS-81-14
November 1981

Abstract

The implementations of most programming languages restrict the maximum size of integers because they are represented in a fixed number of bits. Similarly, reals have a restricted exponent range and a restricted precision. This paper suggests semantics for integer and real types which are closer to the usual mathematical definitions of "integer" and "real", and gives examples to illustrate their use. Possible implementations (both with and without special-purpose hardware) and implications for space and time-efficiency of programs are discussed.

1. Introduction

Because of a desire to implement basic arithmetic operations efficiently on the hardware available, the implementors of most high-level languages have restricted the range of "integers" so that they can be represented in a fixed number of bits (typically 16-64). Similar restrictions have been imposed on both the exponent range and the precision of "reals", which are typically packed into a fixed number (32-96) of bits. When languages have been rigorously defined, the definitions have usually compromised the accepted mathematical meaning of "integer" and "real" by building such restrictions into the definition, although the precise number of bits used is generally left unspecified as an "implementation dependent" detail.

For example, Pascal [A1, J1] has a type "integer" which represents integers in the range -maxint .. +maxint, where "maxint" is an implementation-dependent constant. Pascal has a "real" type, but the standard [A1] does not specify the exponent range or precision. Typical Pascal compilers use the "single-precision" or, in a few cases [S2], the "double-precision" representations supported by the hardware.

Some languages allow several varieties of "reals". For example, Fortran [A2, A3] allows "real" and "double precision", Algol 68 [W1] allows "short real", "real", "long real", "long long real", ... PL/1 [L1], Cobol [A4] and Ada [D1] allow the specification of an "equivalent" number of fixed or floating decimal places. For example, in Ada the user can write:

type F is digits 10;

and the implementation is free to provide a hardware type with at least 10 decimal digits of precision to implement F . The meaning of "at least" is defined in terms of Brown's model [B5] (which models a binary rather than decimal machine). The exponent range can not be specified directly, but the (binary) exponent range is required to be at least four times the number of bits in the floating-point fraction.

The proliferation of implementation-dependent types to approximate the one mathematical concept ("real"), as in Fortran or Algol 68, is unappealing and inhibits portability of programs. The Pascal "solution" of a single type "real" is more elegant, but not useful for solving problems which require more precision or exponent range than is provided by the implementation. Ada, PL/1 and Cobol suffer from a lack of dynamically variable precision. For example, "digits n", where n is a variable, is not permitted in Ada. There are also problems associated with type incompatibilities (especially in Ada [C4, H1]), the decimal specification of a binary representation, the lack of direct control over exponent range, and implementation-dependent restrictions.

In Section 2 we suggest "ideal" semantics for "integer" and "real" types. Some example of program segments which take advantage of these semantics are given in Section 3. In Section 4 we discuss some variations in the possible "ideal" semantics, and in Sections 5 and 6 we suggest some possible implementation techniques. The status of current implementations is outlined in Section 7.

Similar ideas have been suggested and (in some cases) implemented several times before. See, for example, [B2, H3, S1, T1]. Hull's proposals [H2, H3, H4] are discussed briefly in Section 4.

Because of space limitations we do not discuss arithmetic exception handling or the related topics of "infinities" and "NaNs" [K1, S3] here. However, they are important topics which should be considered when a programming language is being designed [F1].

2. Ideal semantics for "integer" and "real" types

There is no doubt about the ideal semantics for type "integer": all integers $\{0, \pm1, \pm2, \ldots\}$ should be represented exactly, and the arithmetic operations + , - , * , mod and div should give exact results. The only implementation-dependent restriction should be that sufficient storage is available. In fact, some symbolic algebra systems [M2] do implement "ideal" integer semantics.

For type "real" some compromise is necessary because most real numbers (e.g. π) can not be represented exactly in any binary or decimal system. The semantics of "real" variables can not be separated from the semantics of the arithmetic operations performed on them.

Before proceeding we need some notation. Let $\beta \geq 2$ be a fixed base (or radix), and $t \geq 1$ a (variable) number of digits which is defined for each arithmetic operation. Let R be the (mathematically defined) set of real numbers. For any $x \in R$ we define

$$N(x) = \begin{cases} -N(-x) & \text{if } x < 0, \\ x-\frac{1}{2} & \text{if } x+\frac{1}{2} \text{ is an } \underline{\text{odd}} \text{ positive integer}, \\ \lfloor x+\frac{1}{2} \rfloor & \text{otherwise}, \end{cases}$$

and

$$r_t(x) = \begin{cases} 0 & \text{if } x = 0 \\ \beta^{k-t} N(\beta^{t-k} x) & \text{if } \beta^{k-1} \le |x| < \beta^k \text{ for integer } k. \end{cases}$$

Thus, $N(x)$ is the "nearest integer to x" (with the "round to even" rule for breaking ties), and $r_t(x)$ is one of the nearest real numbers to x in the set

$$R_t = \{\beta^m j \mid |j| < \beta^t, \ j, m \text{ integer}\}$$

of reals representable exactly with a t-digit, base β fraction.

If p is a variable of type "real", then the "value" of p is either undefined or an exact real number $\bar{p} \in R$. If θ is an arithmetic operation ("+", "−", "*" or "/") occuring in an expression, $\bar{\theta}$ denotes the corresponding (exact) mathematical operation.

We now specify the value $\overline{p\ \theta\ q}$ of an expression $p\ \theta\ q$, where p and q are variables of type "real". Formally,

$$\overline{p\ \theta\ q} = \begin{cases} \text{undefined if } p, q \text{ or } t \text{ is undefined}, \\ \text{undefined if } \theta = \text{"/"} \text{ and } \bar{q} = 0, \\ r_t(r_t(\bar{p})\ \bar{\theta}\ r_t(\bar{q})) \text{ otherwise.} \end{cases} \tag{2.1}$$

Informally, we round the operands p and q to t floating digits, perform the arithmetic operation exactly, then round the result to t digits. This is not difficult if a "sticky bit" is used [K1].

In the following examples we assume the existence of a global variable "precision" which may be copied or changed in the same way as other variables. The value of t is required to satisfy

$$t = \text{digits (precision)} \tag{2.2}$$

where "digits" is some specified positive, monotonic nondecreasing, integer function. The simplest and probably best choice [H3] is $\beta = 10$ and $t = \text{precision}$. However, to allow implementations with $\beta \neq 10$ (e.g. $\beta = 2$) and/or for which t must be a multiple of the wordlength, we only insist on the weaker condition (2.2). For example, we might choose

$$t = 32 \lceil (\log_2 10) * \text{precision}/32 \rceil$$

on a binary machine with a 32-bit word, if "precision" was regarded as specifying a lower bound on the "equivalent" number of decimal places. Alternatively, we might choose t = precision if "precision" was regarded as specifying the exact number of floating (base β) digits. We say that a result is "accurate to precision p" if its relative error is $O(\beta^{-\text{digits}(p)})$.

Lack of space prevents us from discussing "ideal" semantics of real-valued functions such as sqrt, exp, cos here.

3. Some examples

In this section we give some fragments of Pascal-like programs which use the "real" semantics described in Section 2. Several similar examples are given in [H4]. Examples using the "integer" semantics of Section 2 are easy to construct.

a. Inner products

In many linear algebra routines it is desirable to accumulate inner-products in higher precision than the rest of the computation [W2]. Hence, to compute

$$s := \sum_{i=1}^{n} x[i]*y[i]$$

we might have:

```
...
const n = ... ;
s : real;
x, y : array [1..n] of real;
i : 1..n;
...
precision := 2*precision;
s := 0;
for i := 1 to  n  do
    s := s + x[i]*y[i];
precision := precision div 2;
...
```

b. Solution of ill-conditioned problems

Some problems are inherently ill-conditioned and require high-precision computation to obtain moderate-precision solutions. This is justified if the data for the problem is exact, though usually not if the data is contaminated by observational errors (because the ill-condition may magnify these errors to make the "solution" meaningless). As an example we consider the solution of a linear system of equations

$$A\,x = b\ ,$$

where A is nonsingular but ill-conditioned, and A and b are given exactly. The structure of a program which usually gives x accurate to any required precision is:

```
const n = ... ;
p0 : integer;
tol : real;
b, r, x, y : array [1..n] of real;
A, LU : array [1..n, 1..n] of real;
...
precision := ...          {required final precision};
p0 := precision           {save to restore later};
tol := ...                {tol := β^(-digits(p0))};
{set up A and b to current precision}
      repeat
      {compute and save LU decomposition of A with pivoting}
      {compute x using LU decomposition of A}
      precision := 2*precision  {double the precision};
      {recompute A and b to current precision}
      {compute r := Ax-b}
      {solve Ay = r using LU}
      until ||y|| < tol * ||x||;
precision := p0           {restore precision};
```

An alternative termination criterion is $||r|| * \kappa < tol * ||b||$, where κ is an estimate of the condition number of A, obtained as in [C2].

c. Computation of special functions

To compute some mathematically defined function $f(x)$ to precision p by an obvious method (e.g. Taylor's series) often requires working to higher precision [B2, C1]. In order to give a simple example, suppose that

$$f(x) = \sum_{i=0}^{\infty} a_i(x)\ ,$$

where the a_i alternate in sign and $|a_i(x)| > |a_{i+1}(x)|$. The following program computes $f(x)$ to any desired precision.

```
i, p0, del : integer;
a, est, f, tol : real;
...
precision := ...        {required final precision};
tol := ...              {tol := β^-digits(precision)};
p0 := precision;
del := 1                {or some positive function of est/tol};
   repeat
   precision := precision + del   {increase precision};
   f := 0;
   est := 0;
   i := 0;
      repeat            {sum enough terms}
      a := ...     {compute a := a_i(x) accurate to current precision};
      f := f+a;
      est := est + abs(f)  {for rounding error estimate as in [P1]};
      i := i+1
      until abs(a) < tol*abs(f)  {truncation error bounded by abs(a)};
   est := est/β↑digits(precision) {a posteriori rounding error estimate}
   until est < tol*abs(f);
precision := p0       {restore precision};
```

4. Alternative semantics for type real

Hull's proposals [H2, H3, H4] have the same motivation as ours but differ from ours in one major respect: when declared, his "real" variables have some precision associated with them, e.g.

```
                    real (16) x
              or    real (p) y
```

where p may be a variable (as in the declaration of a dynamic array bound). If the "current precision" is higher than the declared precision of the variable on the left of an arithmetic assignment, the result is rounded to the lower precision. With our proposal (described in Section 2) a "real" variable does not have any precision associated with it at the time of its declaration. This seems more natural, as we may not know in advance what precision is required, and at different times the same variable may be used on the left of assignments with varying precisions (as in examples b and c of Section 3).

Hull proposes setting the current precision in a "precision block" declaration. For example, his:

```
begin precision (p)
...              {no jumps out of block}
end
```

is equivalent to our:

```
psave: integer;  {temporary}
   begin
   psave := precision    {save precision};
   precision := p        {set precision to p};
   ...
   precision := psave    {restore precision}
   end
```

We avoided the introduction of "precision blocks" for the following reasons:

1. They introduce a new (and avoidable) syntactic entity.

2. Precision blocks are only possible in block-structured languages, but we planned to incorporate "ideal" real semantics in both block-structured and non-block-structured languages (see Section 7).

Some variations on the definition (2.1) are possible. We might, for example, perform the operation exactly and then round the result to t digits, i.e.

$$\overline{p \ \theta \ q} = r_t(\bar{p} \ \bar{\theta} \ \bar{q}) \tag{4.1}$$

when $\bar{p} \ \bar{\theta} \ \bar{q}$ is defined. This definition was excluded because it would be very expensive to implement in certain cases. For example, in:

```
real p, q, r;
precision := 1000;
p := ... ;
q := ... ;
precision := 10;
r := p/q
```

the division might need to be performed to very high precision. Note, however, that (2.1) and (4.1) are equivalent if p and q were computed with precision less than or equal to the current precision.

5. Possible implementations of type integer

Any implementation of arbitrary-length integers requires some form of dynamic storage management, because the space required to represent the variables and intermediate results in arithmetic expressions is not known in advance. The obvious schemes use

a. Singly linked lists; or
b. Contiguous blocks of storage.

The second scheme has the advantage of permitting faster arithmetic operations (because the arithmetic routines do not have to traverse lists) and has been chosen for the implementation described in Section 7. Scheme (b) can be expected to generate less page faults than scheme (a) on machines with virtual memory. Scheme (a) has the advantage of simplicity; dynamic storage allocation simply requires the maintenance of a free-list, whereas scheme (b) requires an algorithm to allocate and free contiguous blocks of memory efficiently [K2].

With either scheme, arithmetic operations on integers would be much slower than the corresponding operations on one-word integers implemented by hardware or microcode. Hence, when implementing a language similar to Pascal or Ada, it would be important for a compiler to generate single-word instructions where they could be guaranteed to be sufficient. For example, a Pascal-like language might have:

```
const n = 100;
x, y : array [0..n] of integer;
i : 0..n;
...
for i := 0 to n do
   x[i] := x[i div 2] + y[i];
```

Since all indexing operations, do-loop control, etc. require only single-precision operations, only one (possibly) multiple-precision operation needs to be performed in the loop.

On many machines it would be possible to implement operations on arbitrary-precision integers at little cost in space or time if the operands and results happened to be small, although at the expense of processing a memory-protection exception for each "genuine" multiple-precision operation. The idea is that all integers (except those known to be small, as in the example above) would be accessed indirectly through a word which would contain their address (if representable in single-precision) or an illegal address (if not representable in single-precision). The cost for single-precision operations would be one additional level of indirection (and some restrictions on code generation).

For operations involving a multiple-precision operand, the trap handler to which control would pass on an illegal memory reference would have to add a suitable constant to the illegal address (to get the actual address of the block or header of the linked list containing the multiple-precision operand), interpret the instruction which generated the trap (and possibly one or more following instructions), and call the appropriate multiple-precision arithmetic routine. Similar modifications would be required for the trap handler dealing with integer overflow.

On a micro-coded machine, or one designed with multiple-precision arithmetic in mind, it would be possible to reserve one "tag" bit per word as a flag to indicate if the remaining bits were to be regarded (in integer arithmetic operations) as single-precision integers or as pointers to multiple-precision integers. Then the facility for performing multiple-precision arithmetic would not increase the cost of operations on small integers. The cost of performing multiple-precision arithmetic could also be reduced by specially designed hardware [C3, R2].

6. Possible implementations of type real

A pair (e, f) of integers can be used to represent a "real" variable x with value $\beta^e f$ (or, more conventionally, $\beta^{e-t} f$), where $|f| < \beta^t$, β is the base (see Section 2) and t the value of digits (precision) on the last assignment of a value to x. Thus, the problem of implementing type real reduces to the problem of implementing type integer, already discussed in Section 5. For the sake of efficiency in dealing with real variables whose precision matches the precision implemented in floating-point hardware on the machine, ideas similar to those discussed at the end of Section 5 could be used.

In practice it is unlikely that the exponent e of a real variable will exceed the maximum size of a single-precision integer during execution of a correct program. Thus, an implementation might impose this restriction on the allowable exponent range of real variables. This was done in the MP package [B1, B4], for example. A disadvantage is that the possibility of exponent overflow and underflow has to be considered, whereas overflow and underflow are impossible if e is an unrestricted integer.

7. Current and planned implementations

Hull has implemented his proposals for type real (which differ from ours as outlined in Section 4) with both a precompiler [H4] and an Algol-like language [C6]. He is also implementing decimal multiple-precision hardware [C3] so as to make it feasible to choose $\beta = 10$.

We have implemented both type integer and type real (as described in Section 2) by a set of Fortran subroutines which call the MP package [B1, B4] and some Fortran dynamic storage routines [B3]. Using the Augment precompiler [C5], this allows an extension of Fortran incorporating the "ideal" integer and real semantics. The user need not be familiar with the MP package or the actual representation of multiple-precision integer and real variables. The implementation is currently working, but not (as of October 1981) completely tested or documented.

Implementation using Augment and the MP package was considered a "quick and dirty" way of obtaining experience in programming a variety of problems which could benefit from the "ideal" semantics for type integer and real. We plan to implement a modification of Pascal (hardly an extension because no change in the standard Pascal syntax is proposed, only a change in the semantics of type integer and type real). One possibility is to use an existing portable compiler and merely modify the P-code interpreter.

Historically, the project started from an attempt to implement variable-precision interval arithmetic [M1] using Augment and the MP package. We now plan to implement intervals as pairs of "ideal" real variables.

8. Conclusion

In Section 2 we presented "ideal" semantics for type integer and type real. These semantics are actually simpler and more amenable to mathematical analysis (e.g. for correctness proofs) than the usual semantics because they avoid problems of limited range and machine-dependence. The utility of the "ideal" semantics was demonstrated by some examples in Section 3, and alternatives were discussed in Section 4.

The only valid argument against the adoption of our "ideal" semantics (or something similar) is the cost of implementation. In Sections 5 and 6 we argued that this cost need not be very great. A practical demonstration will depend on the success of the implementations described in Section 7.

Acknowledgements

This work stemmed from discussions with several members of IFIP Working Group 2.5; in particular, my debt to Thomas Hull is obvious. The Fortran implementation described in Section 7 was supported by the ARGC.

References

A1. A.M. Addyman, A draft proposal for Pascal, ACM SIGPLAN Notices 15, 4 (April 1980), 1-66.

A2. ANSI, American National Standard Fortran (ANS X3.9-1966), Amer. Nat. Standards Inst., New York, 1966.

A3. ANSI, American National Standard Fortran (ANS X3.9-1978), Amer. Nat. Standards Inst., New York, 1978.

A4. ANSI, American National Standard Cobol (ANS X3.23-1974), Amer. Nat. Standards Inst., New York, 1974.

B1. R.P. Brent, A Fortran multiple-precision arithmetic package, ACM Trans. Math. Software 4 (1978), 57-70. Also ibid 4 (1978), 71-81; ibid 5 (1979), 578-579; ibid 6 (1980), 146-149.

B2. R.P. Brent, Unrestricted algorithms for elementary and special functions, Information Processing 80 (S. Lavington, editor), North-Holland, Amsterdam, 1980, 613-619.

B3 R.P. Brent, Efficient implementation of the first-fit strategy for dynamic storage allocation, Austral. Comp. Sci. Communications 3 (1981), 25-34.

B4. R.P. Brent, MP User's Guide (fourth edition), Report TR-CS-81-08, Dept. of Computer Science, ANU, June 1981, 73 pp.

B5. W.S. Brown, A Simple but Realistic Model of Floating-Point Computation, Computing Science Tech. Report No. 83, Bell Labs, Murray Hill, New Jersey, May 1980 (revised Nov. 1980), 34 pp.

C1. C.W. Clenshaw and F.W.J. Olver, An unrestricted algorithm for the exponential function, SIAM J. Numer. Anal. 17 (1980), 310-331.

C2. A.K. Cline, C.B. Moler, G.W. Stewart and J.H. Wilkinson, An estimate for the condition number, SIAM J. Numer. Anal. 16 (1979), 368-375.

C3. M. Cohen, V.C. Hamacher and T.E. Hull, CADAC: An arithmetic unit for clean arithmetic and controlled precision, Proc. Fifth Symposium on Computer Arithmetic, IEEE Computer Society, 1981, 106-112.

C4. M.G. Cox and S.J. Hammarling, Evaluation of the language Ada for use in numerical computations, Report DNACS 30/80, NPL, Teddington, UK, 1980.

C5. F.D. Crary, A versatile precompiler for nonstandard arithmetics, ACM Trans. Math. Software 5 (1979), 204-217.

C6. A. Curley, PNCL: A Prototype Numerical Computation Language, Tech. Report, Dept. of Computer Science, Univ. of Toronto, to appear.

D1. DoD, Reference Manual for the Ada Programming Language, US Dept. of Defence, July 1980 (reprinted November 1980).

F1. S. Feldman, Language features to support the IEEE floating-point standard, in [R1].

H1. S.J. Hammarling and D.A. Wichmann, Numerical packages in Ada, in [R1].

H2. T.E. Hull, Desirable floating-point and elementary functions for numerical computation, SIGNUM Newsletter 14 (1979), 96-99.

H3. T.E. Hull, The use of controlled precision, in [R1].

H4. T.E. Hull and J.J. Hofbauer, Language Facilities for Multiple Precision Floating Point Computation, with Examples, and the Description of a Preprocessor, Tech. Report 63, Dept. of Computer Science, Univ. of Toronto, 1974, 84 pp.

J1. K. Jensen and N. Wirth, Pascal User Manual and Report (corrected reprint of second edition), Springer-Verlag, Berlin, 1976.

K1. W. Kahan and J. Palmer, On a proposed floating-point standard, SIGNUM Newsletter (Oct. 1979), 13-21.

K2. D.E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, Menlo Park, 1968, Sec. 2.5.

L1. P. Lucas and K. Walk, On the formal description of PL/1, Ann. Rev. Auto. Prog. 6, 3 (1969), 105-182.

M1. R.E. Moore, Methods and Applications of Interval Analysis, SIAM, Philadelphia, 1979.

M2. J. Moses, The evolution of algebraic manipulation algorithms, 1975 Best Computer Papers, I. Auerbach (editor), Petrocelli, 1975, 197-214.

P1. G. Peters and J.H. Wilkinson, Practical problems arising in the solution of polynomial equations, J. Inst. Maths. Applics. 8 (1971), 16-35.

R1. J.K. Reid (editor), The Relationship Between Numerical Computation and Programming Languages, North-Holland, to appear.

R2. R.L. Rivest, A description of a single-chip implementation of the RSA cipher, Lambda, fourth quarter 1980, 14-18.

S1. J.L. Schonfelder and J.T. Thomason, Applications support by direct language extension - an arbitrary precision arithmetic facility in Algol 68, Comp. Centre, Univ. of Birmingham, 1975.

S2. J. Steensgaard-Madsen, H. Snog and M.C. Newey, Pascal 1100 User's Manual, Computer Services Centre, ANU, June 1978, 50 pp.

S3. D. Stevenson, A proposed standard for binary floating-point arithmetic, Draft 8.0 of IEEE Task P754, IEEE Computer 14 (1981), 51-62.

T1. M. Tienari, Varying Length Floating Point Arithmetic: A Necessary Tool for the Numerical Analyst, Tech. Report No. 62, Computer Science Dept., Stanford Univ., 1967.

W1. A. van Wijngaarden, B. Mailloux, J. Peck and C. Kester, Report on the algorithmic language Algol 68, Numerische Mathematik 14, 2 (1969), 79-218.

W2. J.H. Wilkinson, Rounding Errors in Algebraic Processes, HMSO, London, 1963.