9 Real Numbers

9.1 Representations of real numbers

In Chapter 4 we discussed how integer types are used to represent a subset of the mathematical integers. Computation with integer types can cause overflow—a concept which has no meaning for mathematical integers—and the possibility of overflow means that the commutativity and associativity of arithmetical operations is not ensured during the computation.

Representation of real numbers on computers and computation with their representations are extremely problematical, to the point where a specialist should be consulted during the construction of critical programs. This chapter will explore the basic concepts of computation with real numbers; the superficial ease with which real-number computations can be written in a program must not obscure the underlying problems.

First of all let us note that decimal numbers cannot be accurately represented in binary notation. For example, 0.2 (one-fifth) has no exact representation as a binary number, only the repeating binary fraction:

$$0.0011001100110011\ldots$$

There are two solutions to this problem:

- Represent decimal numbers directly, for example by assigning four bits to each decimal numeral. This representation is called *binary-coded decimal* (*BCD*).
- Store binary numbers and accept the fact that some loss of accuracy will occur.

BCD wastes a certain amount of memory because four bits could represent 16 different values and not just the 10 needed for decimals. A more important disadvantage is that the representation is not "natural" and calculation with BCD is much slower than with binary numbers. Thus the discussion will be limited to binary representations; the reader interested in BCD computation is referred to languages such as Cobol which support BCD numbers.

Fixed-point numbers

To simplify the following discussion, it will be given in terms of decimal numbers, but the concepts are identical for binary numbers. Suppose that we can represent seven digits, five before and two after the decimal point, in one 32-bit memory word:

$$12345.67, \quad -1234.56, \quad 0.12$$

This representation is called *fixed-point*. The advantage of fixed-point numbers is that the *accuracy*, that is the absolute error, is fixed. If the above numbers denote dollars and cents, then any error caused by the limited size of a memory word is at most one cent. The disadvantage is that the *precision*, that is the relative error which is the number of significant digits, is variable. The first number uses all available seven digits of precision, while the last uses only two. More importantly, the varying precision means that many important numbers such as the $\$1,532,854.07$ that you won in the lottery, or your \$0.00572 income-tax refund, cannot be represented at all.

Fixed-point numbers are used in applications where total accuracy is essential. For example, accounting computations are usually done in fixed-point, since the required precision is known in advance (say 12 or 16 digits) and the account must balance to the last cent. Fixed-point numbers are also used in control systems where sensors and actuators communicate with the computer in fixed-length words or fields. For example, speed could be represented in a 10-bit field with range 0 to 102.3 km/hour; one bit will represent 0.1 km/hour.

Floating-point numbers

Scientists who have to deal in a wide range of numbers use a convenient notation called *scientific notation*:

$$123.45 \times 10^3, \;\; 1.2345 \times 10^{-8}, \;\; -0.00012345 \times 10^7, \;\; 12345000.0 \times 10^4$$

How can we use this concise notation on a computer? First note that there are three items of information that have to be represented: the *sign*, the *mantissa* (123.45 in the first number) and the *exponent*. On the surface it would seem that there is no advantage to representing a number in scientific notation, because of the varying precisions needed to represent the mantissa: five digits in the first and second numbers above, as opposed to eight digits for the other two.

The solution is to note that trailing zero digits of a mantissa greater than 1.0 (and leading zero digits of a mantissa less than 1.0) can be discarded in favor of changes in the value (not the precision!) of the exponent. More precisely, the mantissa can be multiplied or divided repeatedly by 10 until it is in a form that uses the maximum precision; each such operation requires that the exponent be decremented or incremented by 1, respectively. For example, the last two numbers can be written to use a five-digit mantissa:

$$-0.12345 \times 10^4, \;\; 0.12345 \times 10^{12}$$

It is convenient for computation on a computer if every such number has this standard form, called *normalized* form, where the first non-zero digit is the "tenth's" digit. This also saves on space in the representation, because the decimal point is always in the same position and need not be explicitly represented. The representation is called *floating-point* because we require that the decimal point "float" left or right until the number can be represented with maximum precision.

What is the main disadvantage of computation with floating-point numbers? Consider 0.12345×10^{10} which is normalized floating-point for

$$1,234,500,000$$

and suppose that this is how the bank represented your deposit of

$$\$1,234,567,890$$

Your bank manager would be proud that the *relative error*:

$$\frac{67,890}{1,234,567,890}$$

is a very small fraction of one percent, but you would justifiably demand your $\$67,890$, the *absolute error*.

In scientific calculations relative error is much more important than absolute error. In a program that controls the speed of a rocket, the requirement may be that the error be no more than 0.5% even though this translates into a few km/h during launch and a few hundred km/h nearing orbit. Floating-point computation is much more common than fixed-point, because relative accuracy is required more often than absolute accuracy. For this reason most computers have hardware that directly implements floating-point computation.

Representation of floating-point numbers

Floating-point numbers are stored as *binary* numbers in the normalized form we have described:

$$-0.101100111 \times 2^{15}$$

A typical implementation on a 32-bit computer would assign 1 bit to the sign, 23 bits to the mantissa and 8 bits to the exponent. Since it takes $\log_2 10 \approx 3.3$ bits to store one decimal digit, $23/3.3 \approx 7$ digits of precision are obtained. If more precision is needed, a 64-bit double word with a 52-bit mantissa can achieve about 15 digits of precision.

There is a "trick" that is used to represent more numbers than would otherwise be available. Since all floating-point numbers are normalized, and since the first digit of a normalized mantissa is necessarily 1, this first digit need not be explicitly represented.

The signed exponent is represented by *biasing* the exponent so that it is always positive, and the exponent is placed in the high-order bits of a word next to the sign bit. This makes comparisons easier because ordinary integer comparison may be done without extracting a signed two's-complement exponent field. For example, an 8-bit exponent field with values in the range 0..255 represents exponents in the range $-127..128$ with a bias of 127.

We can now decipher a bit string as a floating point number. The string:

$$1 \;\; 10001000 \;\; 01100000000000000000000$$

is deciphered as follows:

- The sign bit is 1 so the number is negative.
- The exponent is $10001000 = 128 + 8 = 136$. Removing the bias gives

$$136 - 127 = 9$$

- The mantissa is $0.10110\ldots$ (note the hidden bit restored), which is
$$\frac{1}{2}+\frac{1}{8}+\frac{1}{16}=\frac{11}{16}$$
- Thus the number stored is $2^9 \times 11/16 = 352$.

Just as with integers, floating point overflow occurs when the result of the computation is too large:

$$(0.5\times 10^{70})\cdot(0.5\times 10^{80}) = 0.25\times 10^{150}$$

Since the largest exponent that can be represented is 128, the computation overflows.

Consider now the computation:

$$(0.5\times 10^{-70})\cdot(0.5\times 10^{-80}) = 0.25\times 10^{-150}$$

The computation is said to *underflow* because the result is too small to be represented. You may claim that such a number is so small that it might as well be zero, and a computer may choose to treat underflow by storing a zero result, but underflow does represent an error and should be either handled or accounted for.

9.2 Language support for real numbers

All programming languages have support for floating-point calculations. Variables can be declared to be of type float,[1] and floating-point literals are represented using a form similar to scientific notation:

C

```
float f1 = 7.456;
float f2 = -46.64E-3;
```

Note that the literals need not be in binary notation or in normalized form; the conversion is done by the compiler.

A minimum of 32 bits is needed for meaningful floating-point calculations. Often more precision is needed, and languages support declaration and computation with higher precisions. At a minimum *double-precision* variables using 64 bits are supported, and some computers or compilers will support even longer types. Double precision floating-point types are called double in C and Long_Float in Ada.

The notation for double-precision literals varies with the language. Fortran uses a separate notation, replacing the E that precedes the exponent with a D: −45.64D−3. C chooses to store *every* literal in double precision, allowing a suffix F if you want to specify single precision. Be careful if you are storing a large array of floating-point constants.

Ada introduces a new concept, *universal types*, to handle the varying precision of literals. A literal such as 0.2 is stored by the compiler in potentially infinite precision (recall that 0.2 cannot be exactly represented as a binary number). When a literal is actually used, it is converted into a constant of the correct precision:

[1] Pascal and Fortran use the name real.

Ada

```
PI_F:  constant Float      := 3.1415926535;
PI_L:  constant Long_Float := 3.1415926535;
PI:    constant            := 3.1415926535;
F: Float      := PI;       -- Convert number to Float
L: Long_Float := PI;       -- Convert number to Long_Float
```

The first two declarations declare constants of the named types. The third declaration for PI is called a *named number* and is of universal real type. When PI is actually used in the initializations, it is converted to the correct precision.

The four arithmetic operators ($+$, $-$, * and $/$) as well as equality and the relational operators are defined for floating-point types. Mathematical functions such the trigonometric functions may be defined within the language (Fortran and Pascal), or they be supplied by subprogram libraries (C and Ada).

Portability of floating-point

Programs that use floating-point can be difficult to port because different definitions are used for the type specifiers. There is nothing to prevent a compiler for C or Ada from using 64 bits to represent float (Float) and 128 bits to represent double (Long_Float). Both porting directions are problematical: when porting from an implementation which uses a high-precision representation of float to one which uses a low-precision representation, all float's must be converted to double's to retain the same level of precision. When porting from a low-precision to a high-precision implementation, the opposite modification may be needed, because using excess precision wastes execution time and memory.

An elementary partial solution is to declare and use an artificial floating-point type; then only a few lines need be changed when porting the program:

```
typedef double Real;        /* C */
subtype Real is Long_Float;-- Ada
```

See Section 9.4 for Ada's comprehensive solution to the problem of portable computation with real numbers.

Hardware and software floating-point

Our discussion of the representation of floating-point numbers should have made it clear that arithmetic on these values is a complex task. The words must be decomposed, the exponent biases removed, the multiword arithmetic done, the result normalized and the result word composed. Most computers use special hardware for efficient execution of floating-point calculations.

A computer without the appropriate hardware can still execute floating-point calculations using a library of subprograms that *emulate* floating-point instructions. Attempting to execute a floating-point instruction will cause a "bad-instruction" interrupt which will be handled by calling the

appropriate emulation. Needless to say this can be very inefficient, since there is the interrupt and subprogram overhead in addition to the floating-point computation.

If you think that your program may be extensively used on computers without floating-point hardware, it may be prudent to avoid floating-point computation altogether. Instead you can explicitly program fixed-point calculations where needed. For example, a financial program can do all its computation in terms of "cents" rather than fractions of dollars. Of course this runs the risk of overflow if the Integer or Long_Integer types are not represented with sufficient precision.

Mixed arithmetic

Mixed integer and real arithmetic is very common in mathematics: we write $A = 2\pi r$ and not $A = 2.0\pi r$. When computing, mixed integer and floating-point operations must be done with some care. The second form is preferable because 2.0 can be stored directly as a floating-point constant, whereas the literal 2 would have to be converted to a floating representation. While this is usually done automatically by the compiler, it is better to write exactly what you require.

Another potential source of difficulty is the distinction between integer division and floating-point division:

Ada

```
I: Integer := 7;
J: Integer := I / 2;
K: Integer := Integer(Float(I) / 2.0);
```

The expression in the assignment to J calls for integer division; the result, of course, is 3. In the assignment to K, floating-point division is required: the result is 3.5 and it is converted to an integer by rounding to 4.

Languages even disagree on how to convert floating-point values to integer values. The same example in C:

C

```
int i = 7;
int j = i / 2;
int k = (int) ((float i) / 2.0);
```

assigns 3 to both j and k, because the floating-point value 3.5 is truncated rather than rounded!

C implicitly performs mixed arithmetic, converting integers to floating-point types if necessary and converting lower precision to higher precision. Also, values are implicitly converted upon assignment. Thus the above example could be written:

C

```
int k = i / 2.0;
```

While the *promotion* of the integer i to floating-point is clear, programs will be more readable if explicit conversions are used on assignment statements (as opposed to initializations):

```
k = (int) i / 2.0;
```
C

Ada forbids *all* mixed arithmetic; however, any value of a numeric type can be explicitly converted from a value of any other numeric type as shown above.

If efficiency is important, rearrange a mixed expression so that the computation is kept as simple as possible for as long as possible. For example (recalling that literals are considered double in C):

```
int i, j, k, l;
float f = 2.2 * i * j * k * l;
```
C

would be done by converting i to double, performing the multiplication 2.2 * i and so on, with every integer being converted to double. Finally, the result would be converted to float for the assignment. It would be more efficient to write:

```
int i, j, k, l;
float f = 2.2F * (i * j * k * l);
```
C

to ensure that the integer variables are first multiplied using fast integer instructions and that the literal be stored as float and not as double. Of course, these optimizations could introduce integer overflow which would not occur if the computation was done in double precision.

One way to improve the efficiency of any floating-point computation is to arrange the algorithm so that only part of the computation has to be done in double precision. For example, a physics problem can use single precision when computing the motion of two objects that are close to each other (so the distance between them can be precisely represented in relatively few digits); the program can then switch to double precision as the objects get further away from each other.

9.3 The three deadly sins

Every floating-point operation produces a result that is possibly incorrect in the least significant digit because of rounding errors. Programmers writing numerical software should be well-versed in the methods for estimating and controlling these errors. In these paragraphs, we will summarize three serious errors that can occur:

- Negligible addition
- Error magnification
- Loss of significance

Negligible addition occurs when adding or subtracting a number that is very small relative to the first operand. In five-digit decimal arithmetic:

$$0.1234 \times 10^3 + 0.1234 \times 10^{-4} = 0.1234 \times 10^3$$

It is unlikely that your high-school teacher taught you that $x+y=x$ for non-zero y, but that is what has occurred here!

Error magnification is the large absolute error that can occur in floating-point arithmetic even though the relative error is small. It is usually the result of multiplication or division. Consider the computation of $x \cdot x$:

$$0.1234 \times 10^3 \cdot 0.1234 \times 10^3 \; = \; 0.1522 \times 10^5$$

and suppose now that during the calculation of x, there had been a one-digit error, that is an absolute error of 0.1:

$$0.1235 \times 10^3 \cdot 0.1235 \times 10^3 = 0.1525 \times 10^5$$

The error is now 30 which is 300 times the size of the error before the multiplication.

The most serious error is complete loss of significance caused by subtracting nearly equal numbers:

C

```
float f1 = 0.12342;
float f2 = 0.12346;
```

In mathematics, $f2-f1 = 0.00004$ which is certainly representable as a four-digit floating-point number: 0.4000×10^{-4}. However, a program computing f2–f1 in four-digit floating-point will give the answer:

$$0.1235 \times 10^0 \; - \; 0.1234 \times 10^0 \; = \; 0.1000 \times 10^{-3}$$

which is not even close to being a reasonable answer.

Loss of significance is more common than might be supposed because equality calculations are usually implemented by subtracting and then comparing to zero. The following if-statement is thus totally unacceptable:

C

```
f1 = ... ;
f2 = ... ;
if (f1 == f2) ...
```

The most innocent rearrangement of the expressions for f1 and f2, whether done by a programmer or by an optimizer, can cause a different branch of the if-statement to be taken. The correct way of checking equality in floating-point is to introduce a small error term:

C

```
#define Epsilon 10e–20

if ((fabs(f2 – f1)) ¡ Epsilon) ...
```

and then compare the absolute value of the difference to the error term. For the same reason, there is no effective difference between ¡= and ¡ in floating-point calculation.

Errors in floating-point calculations can often be reduced by rearrangement. Since addition associates to the left, the four-digit decimal computation:

$$1234.0 + 0.5678 + 0.5678 = 1234.0$$

is better if done:

$$0.5678 + 0.5678 + 1234.0 = 1235.0$$

to avoid negligible addition.

As another example, the arithmetic identity:

$$(x+y)(x-y) = x^2 - y^2$$

can be used to improve the accuracy of a computation:[2]

Ada

```
X, Y: Float_4;
Z: Float_7;

Z := Float_7( (X+Y)*(X-Y) );          -- This way ?
Z := Float_7( X*X - Y*Y );            -- or this way ?
```

If we let $x = 1234.0$ and $y = 0.6$, the correct value of this expression is 1522755.64. Evaluated to eight digits, the results are:

$$(1234.0 + 0.6) \cdot (1234.0 - 0.6) \;=\; 1235.0 \cdot 1233.0 \;=\; 1522755.0$$

and

$$(1234.0 \cdot 1234.0) - (0.6 \cdot 0.6) \;=\; 1522756.0 - 0.36 \;=\; 1522756.0$$

When $(x+y)(x-y)$ is evaluated, the small error resulting from the addition and subtraction is greatly modified by the magnification caused by the multiplication. By rearranging, the formula $x^2 - y^2$ avoids the negligible addition and subtraction and gives a more correct answer.

9.4 * Real types in Ada

Note: the technical definition of real types was significantly simplified in the revision of Ada from Ada 83 to Ada 95, so if you intend to study the details of this subject, it would be best to skip the older definition.

Floating-types in Ada

In Section 4.6 we described how an integer type can be declared to have a given range, while the implementation is chosen by the compiler:

```
type Altitude is range 0 .. 60000;
```

Similar support for portability of floating-point computations is given by the declaration of arbitrary floating-point types:

[2] Float_4 and Float_7 are assumed to be types with four and seven digit precision, respectively.

```
type F is digits 12;
```

This declaration requests 12 (decimal) digits of precision. On a 32-bit computer this will require double precision, while on a 64-bit computer single precision will suffice. Note that as with integer types, this declaration creates a new type which cannot be used in operations with other types without explicit conversions.

The Ada standard describes in great detail conforming implementations of such a declaration. Programs whose correctness depends only on the requirements of the standard, and not on any quirks of a particular implementation, are assured of easy porting from one Ada compiler to another, even to a compiler on an entirely different computer architecture.

Fixed-point types in Ada

A fixed-point type is declared as follows:

```
type F is delta 0.1 range 0.0 .. 1.0;
```

In addition to a range, when writing a fixed-point type declaration you indicate the absolute error required by writing a fraction after the keyword delta.

Given delta D and a range R, an implementation is required to supply a set of numbers, called *model numbers*, at most D from one another which cover R. On a binary computer, the model numbers would be multiples of a power of two just below the value of D, in this case $1/16 = 0.0625$. The model numbers corresponding to the above declaration are:

$$0,\ 1/16,\ 2/16,\ \ldots,\ 14/16,\ 15/16$$

Note that even though 1.0 is specified as part of the range, that number is not one of the model numbers! The definition only requires that 1.0 be within 0.1 of a model number, and this requirement is fulfilled because $15/16 = 0.9375$ and $1.0 - 0.9375 < 0.1$.

There is a predefined type Duration which is used for measuring time intervals. Fixed-point is appropriate here because time will have an absolute error (say 0.0001 second) depending on the hardware of the computer.

For business data processing, Ada 95 defines *decimal fixed-point types*:

```
type Cost is delta 0.01 digits 10;
```

Unlike ordinary fixed-point types which are represented by powers of two, these numbers are represented by powers of ten and so are suitable for exact decimal arithmetic. The type declared above can hold values up to $99,999,999.99$.

9.5 Exercises

1. What floating-point types exist on your computer? List the range and precision for each one. Is exponent bias used? Normalization? Hidden high-order bit? What infinite or other unusual values exist?

2. Write a program that takes a floating-point number and prints the sign, mantissa and exponent (after removing any bias).
3. Write a program for infinite precision integer addition and multiplication.
4. Write a program to print the binary representation of a decimal fraction.
5. Write a program for BCD arithmetic.
6. Write a program to emulate floating-point addition and multiplication.
7. Declare various fixed-point types in Ada and check how values are represented. How is Duration represented?
8. In Ada there are limitations on fixed-point arithmetic. List and motivate each limitation.