Scientific Computing in the $\mathrm{C^H}$ Programming Language

Dr. Harry H. Cheng, Assistant Professor
Director of Integration Engineering Laboratory
Department of Mechanical & Aeronautical Engineering
and Graduate Group in Computer Science
University of California
Davis, CA 95616
Email: hhcheng@ucdavis.edu
World Wide Web: http://iel.ucdavis.edu

Abstract

We have developed a general-purpose block-structured interpretive programming language. The syntax and semantics of this language called $\mathrm{C^H}$ are similar to C. $\mathrm{C^H}$ retains most features of C from the scientific computing point of view. In this paper, the extension of C to $\mathrm{C^H}$ for numerical computation of real numbers will be described. Metanumbers of -0.0, 0.0, Inf, $-$Inf, and NaN are introduced in $\mathrm{C^H}$. Through these metanumbers, the power of the IEEE-754 arithmetic standard is easily available to the programmer. These metanumbers are extended to commonly used mathematical functions in the spirit of the IEEE 754 standard and ANSI C. The definitions for manipulation of these metanumbers in I/O; arithmetic, relational, and logic operations; and built-in polymorphic mathematical functions are defined. The capabilities of bitwise, assignment, address and indirection, increment and decrement, as well as type conversion operations in ANSI C are extended in $\mathrm{C^H}$. In this paper, mainly new linguistic features of $\mathrm{C^H}$ in comparison to C will be described. Example programs programmed in $\mathrm{C^H}$ with metanumbers and polymorphic mathematical functions will demonstrate capabilities of $\mathrm{C^H}$ in scientific computing.

1 Introduction

We have developed a general-purpose block-structured interpretive programming language. Due to our research interests, this language called $\mathrm{C^H}$ has been developed to be especially suitable for research and applications in scientific and system programming. $\mathrm{C^H}$ is expressive with modern programming constructs and rich sets of data types and operators. At its current implementation, $\mathrm{C^H}$ supports most features of the C programming language except data structures. Some rough edges incompatible with the ANSI C (ANSI, 1989) will be smoothed out in the future. By then, one may consider $\mathrm{C^H}$ as a C language with High-level extensions. $\mathrm{C^H}$ extends the capabilities of C in many aspects. $\mathrm{C^H}$ not only supports C's basic data types such as int and float, but also provides many additional data types such as complex and others. The handling of complex and dual numbers in $\mathrm{C^H}$ are described in (Cheng, 1993, 1993b). The constants, variables, and operators of new data types in $\mathrm{C^H}$ follow the same syntax rules of basic data types such as int and float. C, a modern language originally invented for the Unix system programming (Thompson, 1978; Ritchie and Thompson, 1974), is commonly regarded as a mid-level computer language. $\mathrm{C^H}$ retains low level features of C with respect to interface to hardware. But, $\mathrm{C^H}$ is a high-level language, designed for both novice users and experienced programmers. If one makes mistakes in a C^H program, the system will prompt informative warning or error messages for the debugging of the program.

C^H is a language designed for both scientific and system programming. Currently, Fortran (ANSI, 1978; Cheng, 1989) and C are the two predominant computer languages for scientific computing. C^H has been designed to make the porting of both Fortran and C code to C^H as easy as possible. However, as the name of the language implies, whenever there is a syntax conflict between C and other languages, the interpretation will follow that of C. As a result, the syntax and semantics of C^H are similar to those of C in many aspects. Therefore, unless indicated otherwise, all code fragments included in this paper will have the same implications as those in the ANSI C. The detailed explanations for each single line of code presented in this paper will not be given.

In this paper, the scientific computing aspect of the C^H language will be addressed. C^H retains most features of C from the scientific computing point of view. The ANSI/IEEE 754 standard for binary floating-point arithmetic (IEEE, 1985, Cody et al, 1984) is a significant milestone on the road to consistent floating-point arithmetic with respect to real numbers. This standard has significantly influenced the design of C^H. The IEEE 754 standard distinguishes $+0.0$ from -0.0, which introduces an extra complexity for programming. The rationale for this extra complexity is not well understood and accepted by all computer scientists and C experts (Plauger, 1992). Many have challenged the necessity for the sign of zeros. Apparently, how to handle best "the sign of nothing" is still a topic to be further investigated. Another important feature of the IEEE 754 standard is the internal representation for mathematical infinity and invalid value. The mathematical infinity ∞ is represented by the symbol of Inf. A mathematically indeterminate or an undefined value such as division of zero by zero is represented by NaN which stands for Not-a-Number. Many computer hardware have signed zeros, infinity, and NaN (Motorola, 1989a, 1989b). But, information about low-level and limited high-level instruction sets provided by hardware vendors may not be relevant to the application programmer and most features of a final system depend on the software implementation. Even for IEEE machines, if there is no provision for propagating the sign of zeros, infinity, and NaN in a consistent and useful manner through the software support, they will have to be programmed as if zeros are unsigned without infinity and NaN. For example, the proposed Ada standard does not distinguish -0.0 from 0.0 and has no provision for consistent handling of infinity and NaN (Dritz, 1991a,b,c,d; Hodgson, 1991a,b). As another example, the standard mathematical C library implemented in (Plauger, 1992) has provisions for signed infinities and NaN, but zeros are unsigned. Based on IEEE machines, some vendors provide software support for the IEEE 754 standard through libraries (Sun, 1990a, 1990b; Apple, 1986). However, these special values in libraries are not transparent to the programmer. Due to different design considerations, they have defined different values for many operations and functions discussed in this paper. For example, the SUN's mathematical library will deliver the following results: $\infty^0 = 1; NaN^0 = 1; 0^0 = 1; (-\infty)^x = \infty; (-\infty)^{-\infty} = 0; (-\infty)^0 = 1; (-x)^\infty = \infty; (-x)^{-\infty} = 0; (-\infty)^{-f} = 0$, which differs from C^H. Although the application of symbols such as Inf and NaN can be found in some software packages, their handling of these special numbers is often full of flaws. For example, one can find ComplexInfinity in the software package Mathematica (Wolfram, 1988), and Inf and NaN in MATLAB (MathWorks, 1990). In Mathematica, there is no distinction between complex infinity and real infinities, neither between -0.0 and 0.0; therefore, many operations defined in this paper cannot be achieved in this package. In MATLAB, there is no complex infinity, and one will be surprised by some of its results. At one point, the sign of a zero is honored; but at other point, it may not. For example, according to the IEEE 754 standard, **sqrt**(-0.0) should be -0.0, but, **sqrt**$(-0.0) = 0.0$ in MATLAB (version 4.0, 1992). As another example, **acosh**(Inf) equals NaN whereas **acos**(Inf) is a complex NaN. Results of mathematical functions in many cases are not consistent with mathematical conventions. It is in these grey areas that the standard is not supported in many implementations of hardware and software systems.

To make the power of the IEEE 754 standard easily available to the programmer, the floating-point numbers of -0.0, 0.0, Inf, $-$Inf, and NaN, referred to as *metanumbers*, are introduced in C^H. These metanumbers are transparent to the programmer. Signed zeros +0.0 and -0.0 in C^H behave like correctly signed infinitesimal quantities 0_+ and 0_-; whereas symbols Inf and $-$Inf correspond to mathematical infinities ∞ and $-\infty$, respectively. The manipulation capabilities of Inf and NaN in C^H go way beyond the scope used in mathematical software packages such as Mathematica and MATLAB. The integration of the metanumbers in the C programming language will be described in this paper. The IEEE 754 standard only addresses the arithmetic involving these metanumbers. In this paper, these metanumbers are extended consistently to commonly used mathematical functions in the spirit of the IEEE 754 standard. The linguistic features of C^H, as it is currently implemented, in dealing with metanumbers will be presented in this paper. The emphasis is placed on the handling of metanumbers in I/O; arithmetic, relational, and logic operations; and polymorphic mathematical functions. The concepts presented in this paper have been extended to complex numbers in (Cheng ,1993).

It should be mentioned that related to the work described in this paper is the current effort pursued by Numerical C Extension Group (NCEG), the subcommittee X3J11.1 of the ANSI C X3J11 committee. The NCEG is working on floating-point C extension standard; to make features of the IEEE 754 standard available for use by programmers is one of its efforts. Reviewing its preliminary draft (Thomas, 1993) for the proposed floating-point C extension standard reveals that some features presented in this paper are in conformance with the proposed standard. But, there are many differences between C^H and the proposal. For example, recognizing that operations like `isnan(x)` can be problematic in dealing with NaN, the proposal introduces eight additional relational operators of `!<>=, <>, <>=, !<=, !<. !>=, !>, !<>` on top of the existing operators `<, >, <=, >=, ==, !=`. But, to preserve the clarity and succinctness of C, no additional relational operator has been introduced in C^H. The handling of NaN in C^H will be described in details in this paper. There is no $-$NaN in C^H whereas the sign of NaN is honored in the proposal. The proposal suggests function overloading for elementary mathematical functions in C. However, unlike C++ (Stroustrup, 1987), there is no provision for function overloading in ANSI C. Consequently, mechanisms for function overloading are to be introduced, which will likely complicate the syntax of C. All mathematical functions are built polymorphically with optional auxiliary arguments into C^H itself (Cheng, 1993). Therefore, unlike the proposal, there is no need in C^H for distinction of functions **log**(x) and `log1p`(x) which is expected to be more accurate than **log**() for small magnitude of x because these two functions can be easily reconciled inside C^H. The proposal introduces several new functions, most of these new functions can be easily implemented as external functions in C^H. Due to different considerations, the design of C^H is different from the proposal in some other aspects. For example, C^H is definitive; results of all operations and functions involving metanumbers are properly defined in C^H whereas the proposal still leave rooms for unspecified values. The proposed floating-point C extension is still at its preliminary stage. The final specification and actual language implementation of the proposed standard remain to be seen.

The rest of the paper is arranged as follows. Section 2 presents the number system in C^H. The different data types and their internal memory representations are described. Section 3 describes the external representations of numerical constants in C^H. Section 4 discusses the I/O extension of C to C^H for numerical data and metanumbers. Section 5 defines arithmetic, relational, and logic operations involving metanumbers. In addition, the C^H extensions of bitwise, assignment, address and indirection, increment and decrement operations, as well as explicit type conversions will be highlighted. Section 6 defines polymorphic mathematical functions with metanumbers as input arguments or as returned results. Example programs in section 7 with metanumbers and polymorphic mathematical functions will demonstrate $\mathrm{C^H}$'s capabilities in scientific computing. Some conclusions will be made in section 8.

2 Real Numbers in $\mathrm{C^H}$

$\mathrm{C^H}$ is a loosely typed language. The $\mathrm{C^H}$ programming language has a rich set of data types. Unlike languages such as Pascal (ANSI, 1983) which prohibits automatic type conversion, one data type in $\mathrm{C^H}$ can be automatically converted to another data type if it makes sense in context. As it is currently implemented, $\mathrm{C^H}$ encapsulates Fortran's four numeric data types of integer, real, double precision, and complex. Programming with complex numbers in $\mathrm{C^H}$ will be described in (Cheng, 1993). In this paper, we discuss only the real numbers directly related to scientific computing. The formats of these data stored in a computer memory depends on the machine architecture in use. How these numbers are internally represented in a computer system for manipulation inside $\mathrm{C^H}$ will be illustrated in this section. The discussion is based upon the architecture of the RISC processor for SUN SPARCStations (SUN, 1990b). But, ideas are applicable to all IEEE machines. Data types of short, unsigned, long double, double complex, and long double complex are not available in $\mathrm{C^H}$ at its current implementation, mainly, because our applications of $\mathrm{C^H}$ can bypass these data types. As user's base of $\mathrm{C^H}$ increases, they will be supported in the future if necessary.

2.1 Integers

Integer is a basic data type for any computer languages. An integer in $\mathrm{C^H}$ can be represented in data types of char or int. Numerical manipulations of char and int data in $\mathrm{C^H}$ follow the rules defined in ANSI C.

2.1.1 Char Data Representation

The char data is used to store characters such as letters and punctuations. An array of char can be used to store a string. A character is actually stored in integer according to a certain numerical code such as the ASCII code. Under this code, certain integers represent certain characters. The standard ASCII code ranges from 0 to 127, which can be hold by only 7 bits. In $\mathrm{C^H}$, the char variable is a signed integer ranging from CHAR_MIN to CHAR_MAX. The parameters CHAR_MIN and CHAR_MAX, defined in the ANSI C standard header `limits.h`, are system constants in $\mathrm{C^H}$. Typically, a char constant or variable occupies one byte unit memory. Bit 8 is a sign bit. The maximum positive integer for a signed one-byte representation is 127 or in the binary form of 01111111. A negative number is stored as the binary complement of its absolute value minus 1. For example, the decimal value of -2 is determined by the binary value of 11111110 in a one-byte two's complement value as

$$\mathrm{com}(11111110)_2 = (00000001 + 1)_2 = (10)_2$$

where the subscript of 2 indicates the base of the integer number. The minimum integer values for a signed char is -128 or in binary form of 10000000. The range of integers for a char is then from -128 to $+127$.

2.1.2 Int Data Representation

An int data is a signed integer in $\mathrm{C^H}$. An int number is a whole number which can be negative, positive, or zero. The int ranges from INT_MIN to INT_MAX. The parameters INT_MIN and

INT_MAX, defined in the ANSI C standard header `limits.h`, are precalculated system constants in C^H. Unlike some of C implementations, in which an int data may occupy only two bytes, an int data uses four bytes (32 bits) for storage with one bit for sign in C^H. Negative numbers are stored in four-byte two's complement minus 1. The values of INT_MIN and INT_MAX then become -2147483648 (2^{31}) and 2147483647, respectively. The int type of C^H is the same as the int data type defined in the ANSI C. Operations such as addition, subtraction, multiplication, and division, etc. in C^H are fully compatible with those defined in the ANSI C. For example, the following statements are valid in C^H.

```
char c[2][3], *cptr;
int i, *iptr;            /*  /# comment  */
c[0][1] = 'a';           /# c[0][1] becomes  'a'  */
i = c[0] [ 1];           /# /* i becomes 97, ASCII number for  'a' */
c[1,2] = i+1             /* c[1,2] becomes 'b', ASCII number for 'b' is 98 */
i += c[1 , 2];           /# i becomes 194 = 97 +97
iptr = &i;               /* iptr points to address of i */
*iptr /= 2;              /* i becomes 97 = 194/2 */
```

Like C, comments of a C^H program can be enclosed within a pair of delimiters `/*` and `*/`. These two comment delimiters cannot be nested. In addition, the symbol `/#` in C^H will comment out a subsequent text terminated at the end of a line. A `/#` can be used to comment out `/*` or `*/`, and `/* */` can be used to comment out `/#`. These two companion methods provide a convenient mechanism to comment out a section of C^H code which contains comments. When a comment does not start at the beginning of a line, the use of `/#` is recommended for C^H programs. It should be mentioned that, in ANSI C, a combined use of preprocessor directives `#if`, `#elif`, `#else`, and `#endif` can also comment out a section of C code. Note that arrays in C^H can be declared and accessed by `c[i][j]` or `c[i,j]`. The former is in ANSI C style whereas the later has a Fortran flavor. All white space and tab characters will be ignored in the C^H program, except when they are characters within a string. A program using invisible characters such as a tab character as delimiters and control sequences is very difficult to debug. Such design examples are not difficult to find in computer systems.

2.2 Real Numbers

The integer data type serves well for some software development projects, especially for system programming. However, for scientific computing, the floating-point numbers are used extensively. The floating-point numbers correspond to real numbers which include the numbers between integers. These numbers are defined in C^H as float or double, which are equivalent to real and double precision in Fortran, respectively. Floating-point numbers are analogous to the representations of numbers in scientific notion. Floating-point arithmetic is complicated as compared with the integer arithmetic. This paper mainly addresses issues related to the floating-point operations and built-in functions in C^H.

The most common implementation of floating-point arithmetic is based upon the IEEE 754 standard. In this standard, a float or double is represented in the form of

$$(-1)^{sign} 2^{exponent-bias} 1.f \tag{1}$$

where $1.f$ is the significand and f is the bits in the significand fraction. This normalized float or double number contains a "hidden" bit because it has one more bit of precision than would otherwise be the case.

Table 1: Hexadecimal representation of selected real numbers.

value	float	double
0.0	00000000	0000000000000000
−0.0	80000000	8000000000000000
1.0	3F800000	3FF0000000000000
−1.0	BF800000	BFF0000000000000
2.0	40000000	4000000000000000
−2.0	C0000000	C000000000000000
3.0	40400000	4080000000000000
−3.0	C0400000	C080000000000000
Inf	7F800000	7FF0000000000000
−Inf	FF800000	FFF0000000000000
NaN	7FFFFFFF	7FFFFFFFFFFFFFFF
FLT_MAX	7F7FFFFF	
DBL_MAX		7FEFFFFFFFFFFFFF
FLT_MIN	007FFFFF	
DBL_MIN		000FFFFFFFFFFFFF
FLT_MINIMUM	00000001	
DBL_MINIMUM		0000000000000001

2.2.1 Float Data Representation

The float data type uses 32 bits for its storage. The result of a float data is formulated as

$$(-1)^{sign} 2^{exponent-127} 1.f \tag{2}$$

Bit 31 is a sign bit; it is 1 iff the number is negative. Eight-bit exponent of bits 23-30 is biased by 127; Values of all zeroes and all ones are reserved for metanumbers. Bits 0-22 are the fraction component of a *normalized* significand. The leading integer value 1 of the normalized significand is hidden. The hexadecimal representation of some typical float numbers are given in Table 1. For example, according to formula (2), float numbers 1.0 and -2.0 can be obtained by $(-1)^0 2^{127-127} 1.0 = 1.0$ and $(-1)^1 2^{128-127} 1.0 = 2.0$, respectively. Remember that the fraction of the normalized significand is stored in a binary fraction. The float number 3.0 can be calculated by $(-1)^0 2^{128-127} (1.1)_2 = 2 * (1.1)_2 = 2 * (1.5)_{10} = (3.0)_{10}$ where subscripts indicate the base of the floating-point number. Note that the IEEE 754 standard distinguishes $+0.0$ from -0.0 for floating-point numbers. For user's convenience, these two constants are predefined as system constants Zero and NZero in C^H. NZero stands for Negative-Zero.

The parameter FLT_MAX, defined as the maximum representable finite floating-point number in the float date type in the ANSI C standard header `float.h`, is a precalculated system constant in C^H. As mentioned before that eight-bit exponent of bits 23-30 is biased by 127; values of all ones for eight-bit exponent of bits 23-30 are reserved for metanumbers. If a number is larger than FLT_MAX, which is called an *overflow*, and it will be represented by the symbol of Inf which corresponds to the mathematical infinity ∞. This is the result of many operations such as division of a finite number by zero although an inexact exception may be raised in an IEEE machine. In the same manner, if a number is less than $-$FLT_MAX, it will be represented by $-$Inf which is equivalent to the negative infinity $-\infty$.

The value of the parameter FLT_MIN is defined in the ANSI C standard library header `float.h` as a minimum normalized positive floating-point float number. If a number is less than FLT_MIN, it is called an *underflow.* The IEEE 754 standard provides a *gradual underflow.* When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a denormalized representation. *A denormalized number* is a nonzero number that is not normalized and whose exponent is the minimum exponent fot the storage type. The maximum representable positive denormalized float is defined as FLT_MINIMUM in C^H as shown in Table 1. As one can see that there is only one Unit in the Last Place for FLT_MINIMUM so that it is commonly referred to as *ulp.* Almost all floating-point implementations substitute the value zero for a value which is smaller than FLT_MINIMUM for IEEE machines, FLT_MIN for non-IEEE machines. However, in the arithmetic operations and mathematical functions defined in C^H, there is a qualitative difference between FLT_MINIMUM which is smaller than FLT_MIN and zero. In this paper, by the value of 0.0 means that it is a zero, not a small number. The C^H expressions of 0., 0.00, and .0 are the same as 0.0. In the same token, the following C^H floating-point constant expressions $-0.0, -0., -0.00$, and $-\ .0$ are equivalent. Mathematically, divisions of zero by zero of $\frac{0.0}{0.0}$ and infinity by infinity of $\frac{\infty}{\infty}$ are indeterminate. The results of these operations are represented by the symbol of NaN which stands for Not-a-Number. It should be mentioned that the IEEE 754 standard distinguishes *quiet* NaN from *signaling* NaN. The signaling NaN should generate a signal or raise an exception. In C^H, all NaNs are treated as quiet NaNs. Furthermore, the IEEE 754 standard does not interpret the sign of NaN. However, many floating-point arithmetic implementations such as in the SUN's ANSI C, Apple's Standard Apple Numeric Environment, and preliminary proposed floating-point C extensions distinct NaN from $-$NaN. But, from the user's point of view, what is the difference between a negative Not-a-Number and a positive Not-a-Number? After all, Not-a-Number is not a number. Therefore, no $-$NaN will be produced as a resultant of arithmetic and functions in C^H although it can be created by manipulating the bit pattern of the memory location of a float variable. The expression $-$NaN is interpreted as NaN in C^H. The metanumbers are treated just as regular floating-point numbers. The internal hexadecimal representations of the metanumbers for the float type are also given in Table 1.

2.2.2 Double Data Representation

For a large range of representable floating-point numbers, a double data can be used in C^H. The double data type uses 64 bits as its storage. The result of the double data is formulated as

$$(-1)^{sign} 2^{exponent-1023} 1.f \tag{3}$$

Bit 63 is a sign bit; it is 1 iff the number is negative. Eleven-bit exponent of bits 52-62 is biased by 1023; values of all zeroes and all ones are reserved for metanumbers. Bits 0-51 are fractional component of normalized significand. Like float, the integral value 1 of the normalized significand is hidden. The hexadecimal representation of some typical double numbers are also given in Table 1. Note that the width and bias value of the exponent of double is different from those of float. Therefore, a float cannot be converted into a double just by padding zeroes in its fraction. On the other hand, when a double data is cast into a float, the result cannot be obtained just by ignoring the values in bits 0-31. Note that there is no external distinction between float Inf and double Inf although their internal representations differ. This is also true for metanumbers $-$Inf and NaN. Similar to float, parameters DBL_MAX, DBL_MIN, and DBL_MINIMUM are precalculated constants in C^H. The internal memory representations of these special finite double floating-point numbers are also given in Table 1. Note that due to the finite precision of the floating-point number representation, the exact values of irrational numbers such as π are not representable in a computer system whether they are represented in float or double.

3 Constants of Real Numbers

In this section, we will describe the external representations of data types discussed in the previous section. Besides declared variables and system defined parameters, all different data types in C^H can have their corresponding constants at the programmer's disposal.

Char and int constants in C^H are in full compliance with the ANSI C standard. A character constant, stored as an integer, can be written as one character within a pair of single quotes like `'x'`. Character constants enclosed in a pair of single quotes cannot contain the ' character. In order to represent the ' character and certain other characters such as a newline character, the escape sequence may be used. For example, `c = '\''` will assign the ' character to c while `c = '\n'` will give c a newline character.

A decimal integer constant like 12345 is an int. An integer can also be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant indicates an octal integer whereas a leading 0x or 0X means hexadecimal. Besides these integral values defined in ANSI C, C^H introduces a binary constant with leading 0b or 0B. For example, decimal 30 can be written as 036 in octal, 0X1e or 0x1E in hexadecimal, and 0b11110 or 0B11110 in binary. Note that expressions like 029 and 0b211 are illegal, which can be detected by C^H.

The value of 0 in C^H means that it is an integer zero. Unlike real numbers, there is no 0_- in int. Therefore, the integer value of -0 equals 0 in C^H. The domain [$-$FLT_MAX, FLT_MAX] of real numbers is larger than the domain [$-$INT_MIN, INT_MAX] of integer numbers. When a real number which is smaller than INT_MIN, including $-$Inf, is converted to an integer, the result is INT_MIN. For a real number which is larger than INT_MAX, including Inf, the converted integral value is INT_MAX. When NaN is assigned to an integral variable, the system will print a warning message, and the resultant integral value becomes INT_MAX whose memory map is the same as that of NaN.

In K&R C (Kernighan and Ritchie, 1978; Ritchie et al, 1978) all floats in expressions are converted into doubles before evaluation. As a result, any operations involving floating-point operands, even with two float operands, will produce a double result. This is not applicable to many scientific computations in which speed and memory of a program are very critical. The inconvenient floating-point operation modes for 32-bit operands and 64-bit operands of the original hardware platform, a PDP-11/45 FPP, for running C programs was a major factor in the design of this implicit data conversion of K&R C (Plauger, 1992; Rosler, 1984). Although this indiscriminate conversion is sometimes complemented with a positive tone for its generosity, it is harshly criticized by the numerically-oriented scientific programmers as a language design fault (Press, et al, 1990). Because of the indiscriminate conversion rules in the early design of C, every floating point constant like 3.5 and 3e7 is taken as double. This default double mode for floating-point constants has been carried over to the ANSI C standard. However, the ANSI C has provided a mechanism to specify a float constant. The suffixes F or f indicate a float constant.

In regards to the default data type of floating-point constants, C^H follows the lead of Fortran, but with an ANSI C modern syntax style. Floating-point numbers are represented in scientific notation. All floating-point constants such as $2.4, 2e+3, -2.E-3$, and $+2.1e3$ are float constants by default because, in most applications, a floating-point constant with a small number of digits after a decimal point is intended to be float. This default mode, however, can be switched by the function **floatconst**(onoff). After execution of command **floatconst**(FALSE), the aforementioned floating-point constants will be taken as double. However, the default mode can always be overruled by the suffixes F or f for float, D or d for double. For example, constants $3.4e3F, 3E - 3f$, and $3e + 3F$ are floats whereas constants $3.4e3D, 3E - 3d$, and $3e + 3D$ are doubles regardless of the default mode for floating-point constants. But, the constant metanumbers Zero, NZero, ±Inf, and NaN are always taken as floats unless they are values of double variables. According to this design, the range of representable floating-point number can be expanded automatically. For example, the values of FLT_MAX and DBL_MAX for SUN SPARCStations are $3.4e38$ and $1.8e308$, respectively. The following C$^{\mathrm{H}}$ program

```
printf("pow(10.0, 39) < Inf is %d \n", pow(10.0, 39) < Inf);
floatconst(FALSE);
printf("pow(10.0, 39) < Inf is %d \n", pow(10.0, 39) < Inf);
```

will print out
pow(10.0, 39) < Inf is 0
pow(10.0, 39) < Inf is 1
In the first statement of the program, the value of 10^{39} calculated by **pow**(10.0, 39) has overflowed as Inf because it is larger than FLT_MAX. By switching the default mode of floating-point constants to double through the function **floatconst**(FALSE), the value of 10^{39} calculated by **pow**(10.0, 39) in double data is still within the representable range of $-$DBL_MAX < **pow**(10.0,39) < DBL_MAX. In the second case, the metanumber Inf is expanded as a double infinity larger than DBL_MAX. The float mode for floating-point constants can be switched back by the command **floatconst**(TRUE). With this mode switching function, both Fortran and C code can be ported to C$^{\mathrm{H}}$ conveniently. Details about relational operator < and polymorphic function **pow**() will be discussed in sections 5 and 6, respectively. In the remaining presentation of this paper, we assume that the default mode for floating-point constants is float.

4 I/O for Real Numbers

In ANSI C, the input of integers and floating-point numbers are obtained through the standard I/O functions **scanf**(), **fscanf**(), and etc.; the output is accomplished using the function **printf**(), **fprintf**(), and etc.. These functions are also available in C$^{\mathrm{H}}$ and will be in full compliance with the ANSI C standard. However, implementation of these functions in C$^{\mathrm{H}}$ is different from C. In this section, the differences of these functions between C$^{\mathrm{H}}$ and C, and enhancements of these functions in C$^{\mathrm{H}}$ will be discussed.

The major difference of these functions between C$^{\mathrm{H}}$ and C is that these functions are built-in internal functions in C$^{\mathrm{H}}$ whereas they are external functions in C. Therefore, they can be reconciled inside C$^{\mathrm{H}}$ so that they are more flexible and powerful. The standard input/output/error devices `stdin/stdout/stderr` defined in the ANSI C header `stdio.h` are provided as system constants in C$^{\mathrm{H}}$. The inclusion of header `stdio.h` in a program is, therefore, unnecessary in C$^{\mathrm{H}}$. Other than this difference, a C programmer will not notice any difference in these functions between C$^{\mathrm{H}}$ and C. But, these I/O functions in C$^{\mathrm{H}}$ are enhanced. Here, we only briefly discuss the enhancements related to real numbers for the function **printf**(). The underlining principle can be applied to other I/O functions as well. The format of function **printf**() in C$^{\mathrm{H}}$ is as follows

```
int printf(char *format, arg1, arg2, ...)
```

The function **printf**() prints output to the standard output device under the control of the string pointed to by `format` and returns the number of characters printed. If the format string contains two types of objects: ordinary characters and conversion specifications beginning with a character of % and ending with a conversion character, the ANSI C rules for **printf**() will be used. Besides the control characters specified by the ANSI C standard, $\mathrm{C^H}$ has one more conversion character 'b' which is used to print real numbers in binary format. An integer number between the symbol % and the character 'b' specifies how many bits starting with bit 0 shall be printed. If without an integer number between the symbol % and the character 'b', the default format will print int data without leading zeroes, float data in 32 bits, and double data in 64 bits. This binary format is very convenient to examine the bit patterns of metanumbers. If the format string in **printf**() contains only ordinary characters, the subsequent numerical constants or variables will be printed according to preset default formats. The default format for int, float, and double are %d, %f, and %lf, respectively. The metanumbers Inf and NaN are treated as regular numbers in I/O functions. The default data types for these numbers are float. The following $\mathrm{C^H}$ program illustrates how b-format and metanumbers are handled by the I/O functions **printf**() and **scanf**().

```
float fInf, fNaN;
double dInf, dNaN;
printf("Please type 'Inf NaN Inf NaN' \n");
scanf(&fInf, &fNaN, &dInf, &dNaN);
printf("The float   Inf = %f\n", fInf);
printf("The float  -Inf = ", -fInf, "\n");
printf("The float   NaN = %f\n", fNaN);
printf("The float   Inf = %b \n", fInf);
printf("The float  -Inf = %b \n", -fInf);
printf("The float   NaN = %b \n", fNaN);
printf("The double  Inf = %lf\n", dInf);
printf("The double -Inf = ", -dInf, "\n");
printf("The double  NaN = %lf\n", dNaN);
printf("The double  Inf = %b \n", dInf);
printf("The double -Inf = %b \n", -dInf);
printf("The double  NaN = %b \n", dNaN);
printf("The int     2   = %b \n",  2);
printf("The int     2   = %32b \n",  2);
printf("The int    -2   = %b \n", -2);
printf("The float   0.0 = %b \n", 0.0);
printf("The float  -0.0 = %b \n", -0.0);
printf("The float   1.0 = %b \n",  1.0);
printf("The float  -1.0 = %b \n", -1.0);
printf("The float   2.0 = %b \n",  2.0);
printf("The float  -2.0 = %b \n", -2.0);
```

The first two lines of the program declares two float variables fInf and fNaN, and two double variables dInf and dNaN. The function **scanf**() will get Inf and NaN for the declared variables from the standard input device which is the terminal keyboard in this example. These metanumbers will be printed in default formats %f for float and %lf for double. These numbers are also printed using the binary format %b. For comparison, the memory storage for integers of ± 2, and floats of $\pm 0.0, \pm 1.0, \pm 2.0$ are printed. The result of the interactive execution of the above program is shown as follows

```
Please type 'Inf NaN Inf NaN'
```

Inf Nan Inf Nan

```
The float   Inf = Inf
The float  -Inf = -Inf
The float   NaN = NaN
The float   Inf = 01111111100000000000000000000000
The float  -Inf = 11111111100000000000000000000000
The float   NaN = 01111111111111111111111111111111
The double  Inf = Inf
The double -Inf = -Inf
The double  NaN = NaN
The double  Inf = 0111111111110000000000000000000000000000000000000000000000000000
The double -Inf = 1111111111110000000000000000000000000000000000000000000000000000
The double  NaN = 0111111111111111111111111111111111111111111111111111111111111111
The int     2   = 10
The int     2   = 00000000000000000000000000000010
The int    -2   = 11111111111111111111111111111110
The float   0.0 = 00000000000000000000000000000000
The float  -0.0 = 10000000000000000000000000000000
The float   1.0 = 00111111100000000000000000000000
The float  -1.0 = 10111111100000000000000000000000
The float   2.0 = 01000000000000000000000000000000
The float  -2.0 = 11000000000000000000000000000000
```

where the second line in italic is the input and the rest are the output of the program. As one can see that, for metanumbers Inf, −Inf, and NaN, there is no difference between float and double types from the user's point of view. It can be easily verified that the bit-mappings of all these numbers in memory match with data representations discussed in the previous sections.

5 Real Operations

In this section, the arithmetic, relational, logic, bitwise, assignment, address and indirection, increment and decrement operations, as well as explicit type conversions of real numbers in C^H will be discussed. The operation precedence for different operators in C^H are in full compliance with the ANSI C standard, except the new operator `^^` introduced in section 5.2. Following the ANSI C standard, the algorithms and resultant data types of operations for floating-point numbers will depend on the data types of operands in C^H. The conversion rules for char, int, float, and double in C^H follows the type conversion rules defined in the ANSI C standard. A data type that occupies less memory can be converted to a data type that occupies more memory space without loss of any information. For example, a char integer can be cast into int or float without problem. However, a reverse conversion may result in loss of information. The order of real numbers in C^H ranges from char, int, float, to double. The char data type is the lowest and double the highest. Like the ANSI C, the algorithms and resultant data types of the operations depend on the data types of operands in C^H. For binary operations, such as addition, subtraction, multiplication, and division, the resultant data type will take the higher order data type of two operands. For example, addition of two float numbers will result in a float number while addition of a float number and a double number will become a double number.

The operation rules for regular real numbers and metanumbers in C^H are presented in Tables 2 to 12. In Tables 2 to 12, x, x1, and x2 are regular positive normalized floating-point numbers in float or double; metanumbers 0.0, −0.0, Inf, −Inf, and NaN are constants or the values of float or

Table 2: Negation results.

Negation −							
operand	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
result	Inf	x1	0.0	−0.0	−x2	−Inf	NaN

Table 3: Addition results.

Addition +							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	Inf	Inf	Inf	Inf	Inf	NaN
y2	−Inf	y2−x1	y2	y2	y2+x2	Inf	NaN
0.0	−Inf	−x1	0.0	0.0	x2	Inf	NaN
−0.0	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
−y1	−Inf	−y1−x1	−y1	−y1	−y1+x2	Inf	NaN
−Inf	−Inf	−Inf	−Inf	−Inf	−Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

double variables. By default, the constant metanumbers are float constants.

Table 4: Subtraction results.

Subtraction −							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	Inf	Inf	Inf	Inf	Inf	NaN	NaN
y2	Inf	y2+x1	y2	y2	y2−x2	−Inf	NaN
0.0	Inf	x1	0.0	0.0	−x2	−Inf	NaN
−0.0	Inf	x1	0.0	−0.0	−x2	−Inf	NaN
−y1	Inf	−y1+x1	−y1	−y1	−y1−x2	−Inf	NaN
−Inf	NaN	−Inf	−Inf	−Inf	−Inf	−Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

5.1 Arithmetic Operations

For the negation operation shown in Table 2, the data type of the result is the same as the data type of the operand, a real number will change its sign by negation operation. There is no −NaN in C^{H}. The leading plus sign '+', an unary plus operator, in an expression such as $+57864 - x$ will be ignored. It should be pointed out that the negation of a positive integer zero is still a positive zero. Based upon two's complement representation of negative integer numbers discussed before, we cannot represent Inf and NaN in the int data type.

Note that, according to the IEEE 754 standard, some operations depend on the rounding mode. For example, in case of rounding toward zero, overflow will deliver FLT_MAX rather than Inf with

Table 5: Multiplication results.

Multiplication $*$							
left operand	right operand						
	$-$Inf	$-$x1	$-$0.0	0.0	x2	Inf	NaN
Inf	$-$Inf	$-$Inf	NaN	NaN	Inf	Inf	NaN
y2	$-$Inf	$-$y2$*$x1	$-$0.0	0.0	y2$*$x2	Inf	NaN
0.0	NaN	$-$0.0	$-$0.0	0.0	0.0	NaN	NaN
$-$0.0	NaN	0.0	0.0	$-$0.0	$-$0.0	NaN	NaN
$-$y1	Inf	y1$*$x1	0.0	$-$0.0	$-$y1$*$x2	$-$Inf	NaN
$-$Inf	Inf	Inf	NaN	NaN	$-$Inf	$-$Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 6: Division results.

Division $\div$							
left operand	right operand						
	$-$Inf	$-$x1	$-$0.0	0.0	x2	Inf	NaN
Inf	NaN	$-$Inf	NaN	NaN	Inf	NaN	NaN
y2	$-$0.0	$-$y2/x1	$-$Inf	Inf	y2/x2	0.0	NaN
0.0	$-$0.0	$-$0.0	NaN	NaN	0.0	0.0	NaN
$-$0.0	0.0	0.0	NaN	NaN	$-$0.0	$-$0.0	NaN
$-$y1	0.0	y1/x1	Inf	$-$Inf	$-$y1/x2	$-$0.0	NaN
$-$Inf	NaN	Inf	Inf	$-$Inf	$-$Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

the appropriate sign. This rounding mode is necessary for Fortran implementation and for machines that lack infinity. If the rounding mode is round toward $-\infty$, both $-0.0+0.0$ and $0.0-0.0$ deliver -0.0 rather than 0.0. For scientific programming, consistency and determinacy are essential. C^H is currently implemented using the default rounding mode of round to nearest so that overflow will result in Inf, and both $-0.0+0.0$ and $0.0-0.0$ deliver 0.0 as shown in Tables 3 and 4. Note that the modulus operator % in C^H is ANSI C compatible.

For addition, subtraction, multiplication, and division operations shown in Tables 3 to 6, the resultant data type will be double if any one of two operands is double; otherwise, the result is a float. Notice that the mathematically indeterminate expressions such as $\infty-\infty, \infty*0.0, \frac{\infty}{\infty}$, and $\frac{0.0}{0.0}$ will result in NaNs. The values of ± 0.0 play important roles in the multiplication and division operations. For example, a finite positive value of x2 divided by 0.0 results in a positive infinity $+\infty$ while division by -0.0 will create a negative infinity $-\infty$. If any one of operands of binary arithmetic operations is NaN, the result is NaN.

5.2 Relational Operations

For relational operations given in Tables 7-12, the result is always an integer with a logic value of 1 or 0 corresponding to TRUE or FALSE, which are predefined system constants. According to the IEEE 754 standard, there is a distinction between +0.0 and -0.0 for floating-point numbers. In C^H, the value of 0.0 means that the value approaches zero from positive numbers along the real line and it is a zero; the value of -0.0 means that the value approaches zero from negative numbers along the real line and it is infinitely smaller than 0.0 in many cases. Signed zeros +0.0 and -0.0 in a C^H program behave like correctly signed infinitesimal quantities 0_+ and 0_-, respectively. Although there is a distinction between -0.0 and 0.0 for floating-point numbers in many operations, according to the IEEE 754 standard, the comparison shall ignore the sign of zeros so that -0.0 equals 0.0 in relational operations. For the convenience of the programmer, two polymorphic logic operations `isposzero(x)` and `isnegzero(x)` are introduced in C^H, which can test if the argument x is 0.0 or -0.0. The argument x can be char, int, float, or double. If x is 0.0, `isposzero(x)` and `isnegzero(x)` return 1 and 0, respectively. If x is -0.0, `isposzero(x)` and `isnegzero(x)` return 0 and 1, respectively. If x is a complex or dual number, only its real part will be used in these operations. More elaborative, but less frequently used, functions such as `signbit(x)` and `copysign(x,y)` can be easily implemented as external functions in C^H. The value of -0.0 could be regarded different from 0.0 for comparison operations in C^H. But, for the convenience of porting C code to C^H, zero is unsigned in comparison operations. The equality for metanumbers have different implications in C^H. Two identical metanumbers are considered being equal to each other. As a result, comparing two Infs or two NaNs will get logic TRUE. This is just for the convenience of programming because, mathematically, the infinity of ∞ and Not-a-Number of NaN are undefined values which cannot be compared with each other. Metanumbers of Inf, $-$Inf, and NaN in C^H are treated as regular floating-point numbers consistently in arithmetic, relational, and logic operations. There is no need to use functions such as `isnan(x), isinf(x)`, etc. as is introduced in some software packages and mathematical libraries according to the recommendation of the IEEE 754 standard. Note that NaN is unordered and does not compare equal to itself in the IEEE 754 standard. However, for the convenience of the programmer, NaN is handled in the same manner as Inf in C^H. NaN is still unordered, but it equals itself, which is the only place in which C^H is not in compliance with the IEEE 754 standard. The difference from the standard is likely to cause arguments and resistances. However, with this slight change, programming with metanumbers is much cleaner than would otherwise be the case.

Table 7: Less than comparison results.

Less than comparison <							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	0	0
y2	0	0	0	0	y2 < x2	1	0
0.0	0	0	0	0	1	1	0
−0.0	0	0	0	0	1	1	0
−y1	0	−y1 < −x1	1	1	1	1	0
−Inf	0	1	1	1	1	1	0
NaN	0	0	0	0	0	0	0

Table 8: Less than or equal comparison results.

Less or equal comparison <=							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 <= x2	1	0
0.0	0	0	1	1	1	1	0
−0.0	0	0	1	1	1	1	0
−y1	0	−y1 <= −x1	1	1	1	1	0
−Inf	1	1	1	1	1	1	0
NaN	0	0	0	0	0	0	1

Table 9: Equal comparison results.

Equal comparison ==							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 == x2	0	0
0.0	0	0	1	1	0	0	0
−0.0	0	0	1	1	0	0	0
−y1	0	−y1 == −x1	0	0	0	0	0
−Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	1

Table 10: Greater than or equal comparison results.

Greater or equal comparison >=							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	1	0
y2	1	1	1	1	y2 >= x2	0	0
0.0	1	1	1	1	0	0	0
−0.0	1	1	1	1	0	0	0
−y1	1	−y1 >= −x1	0	0	0	0	0
−Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	1

Table 11: Greater than comparison results.

Greater than comparison >							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	0
y2	1	1	1	1	y2 > x2	0	0
0.0	1	1	0	0	0	0	0
−0.0	1	1	0	0	0	0	0
−y1	1	−y1 > −x1	0	0	0	0	0
−Inf	0	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

Table 12: Not equal comparison results.

Not equal comparison !=							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	1
y2	1	1	1	1	y2 != x2	1	1
0.0	1	1	0	0	1	1	1
−0.0	1	1	0	0	1	1	1
−y1	1	−y1 != −x1	1	1	1	1	1
−Inf	0	1	1	1	1	1	1
NaN	1	1	1	1	1	1	0

5.3 Logic Operations

In C^H, there are four logic operators `!`, `&&`, `||`, and `^^` corresponding to logic operations `not`, `and`, `inclusive or`, and `exclusive or`, respectively. The operations of `!`, `||`, `&&` in C^H comply with the ANSI C standard. The operator `^^` is introduced in C^H due to the consideration of programming convenience and orthogonality between logic operators and bitwise operators. Note that, like ANSI C, both the `&&` and `||` operations in C^H permit the right operand to be evaluated only if the left operand evaluates to TRUE and FALSE, respectively. This "short circuit" behavior for the `^^` operator does not exist because, for either TRUE or FALSE of the first operand, an exclusive-or operation can return TRUE, depending on the second operand. The precedence of operator `^^` is higher than operator `||`, but lower than `&&`. This operation precedence is similar to that for bitwise operators `&`, `|`, and `^`, which will be discussed in the next section. Because there are only two values of either TRUE or FALSE for logic operations, the values of ± 0.0 are treated as logic FALSE while the metanumbers $-$Inf, Inf, and NaN are considered as logic TRUE. For example, evaluations of $!(-0.0)$ and !NaN will get the values of 1 and 0, respectively.

5.4 Bitwise Operations

In C^H, there are six bitwise operators `&`, `|`, `^`, `<<`, `>>`, and `~`, corresponding to `bitwise and`, `inclusive or`, `exclusive or`, `left shift`, `right shift`, and `one's complement`, respectively. These operators in C^H are in full compliance with the ANSI C standard. They can only be applied to integral data which are char and int at its current implementation of C^H. The returned data type depends on the data types of operands. The result of the unary operator `~` keeps the data type of its operand. Results of binary operators `&`, `|`, and `^` will have the higher data type of two operands. The binary operators `<<` and `>>` return the data type of the left operand.

However, some undefined behaviors in ANSI C are defined in C^H. For operators `<<` and `>>`, the right operand can be any data type so long it can be converted into int internally whereas the right operand must be a positive integral value in ANSI C. In C^H, if the right operand is negative integral value which may be converted from a floating-point data, the shifting direction will be reversed. For example, the expression of 7 `<<` -2.0 is equivalent to 7 `>>` 2.0 in C^H. Therefore, only one of these two shift operators is needed in C^H. The use of operator `<<` is recommended for C^H programming. A program with dual shift directions for one operator can be cleaner as compared with unidirectional shifts of two operators.

5.5 Assignment Operations

Besides the regular assignment statement, there are nine assignment operators of `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `<<=`, and `>>=`. These assignment operators are ANSI C compatible. A *lvalue* is any object that occurs on the left hand side of an assignment statement. The lvalue refers to a memory such as a variable or pointer, not a function or constant. The C^H expression of `lvalue op= rvalue` is defined as `lvalue = lvalue op rvalue` where `lvalue` is any valid lvalue including complex numbers discussed in (Cheng, 1993) and it is only evaluated once. For example, `i += 3` is equivalent to `i = i+3`, and **`real`**`(c) *= 2` is the same as `real(c) = real(c)*2`. But, statement `*ptr++ += 2` is different from statement `*ptr++ = *ptr++ +2` because lvalue `*ptr++` contains an increment operation. The operation rules for operators of `+`, `-`, `*`, `/`, `&`, `|`, `^`, `<<`, and `>>` have been discussed in the previous sections.

5.6 Address and Indirection Operations

The unary operator `&` gives the address of an object. The operator `&`, which is ANSI C compatible, can only be applied to a valid lvalue.

When an unary indirection operator `*` is applied to a pointer, it accesses the object the pointer points to. A pointer and an integer can be added or subtracted. Th expression `ptr+n` gives the address of the n-th object beyond the one `ptr` currently points to. The memory locations of pointers `ptr+n` and `ptr` are `n*sizeof(*ptr)` bytes apart, i.e., n is scaled to `n*sizeof(*ptr)` bytes according to declaration of pointer variable `ptr`. Pointer subtraction for pointers with the same data type is permitted. If `ptr1` $>$ `ptr2, ptr1` $-$ `ptr2i` gives the number of objects between `ptr2` and `ptr1`. Array of pointers can also be declared. When a pointer is declared, it is initialized to zero. The symbolic constant NULL, instead of zero, can be used in the program. If `ptr` is NULL; the operand `*ptr` in an expression is evaluated as zero, when `*ptr` is used as a lvalue, a memory of **sizeof**(*ptr) will be allocated automatically for pointer `ptr`. In both cases, the system will print out warning messages. The automatical memory allocation for a pointer which does not point to a valid location can avoid a system crash.

Two pointers and constant NULL can be used in the relational operations `<, <=, ==, >=, >,` and `!=`. In assignment and relational operations, pointers with different data types can work together without explicit type conversions. For example, following are the valid C^H program.

```
int *iptr;
float *fptr;
iptr = (int *)malloc(90);
fptr = malloc(80); /# fptr = (float *)malloc(80)
if(iptr !=  NULL && iptr != fptr)
  free(iptr);
iptr = fptr;
```

Unlike ANSI C, not only all variables are initialized to zero when they are declared, but also the memory allocated by either function **malloc**() or **calloc**() is initialized to zero in C^H. The casting operation for three memory allocation functions **malloc**(), **calloc**(), and **realloc**() is unnecessary. If no memory is available, these functions will return NULL and the system will print out error messages. The function **free**(ptr) will deallocate the memory allocated by these three functions and set pointer `ptr` to NULL. In C, `ptr` is not set to NULL when the memory it points to is deallocated. This dangling memory makes the debugging of the C program very difficult because the problem will not surface until this deallocated memory is claimed again by other parts of the program. The other related functions such as **memcpy**() in C^H for memory manipulations are ANSI C compatible.

As described before, there are several system defined parameters such as NaN, Inf, FLT_MAX, INT_MIN, FLT_EPSILON, etc.. These parameters cannot be used as lvalues so that an accidental change of values of these parameters can be avoided. However, if really necessary, the values of these parameters can be modified by accessing their memory locations through pointers. For example, a numerical algorithm may depend on the parameters FLT_EPSILON and Inf. One can change the values of FLT_EPSILON to 10^{-4} and Inf to FLT_MAX by the following C^H code

```
float *fptr;
fptr = & FLT_EPSILON; *fptr = 1e-4;
fptr = &Inf; *fptr = FLT_MAX;
```

which may, in effect, change the underlying numerical algorithm.

5.7 Increment and Decrement Operations

C is well-known for the succinctness of its syntax. The increment operator `++` and decrement operator `--` are unique to C. These two operators in C^H are compatible with ANSI C. The increment operator `++` adds 1 to its operand whereas the decrement operator `--` subtracts 1. If `++` or `--` is used as a prefix operator, the expression increments or decrements operand before its value is used, respectively. If it is used as a postfix operator, the operation will be performed after its value has been used.

However, additional functions are added to these two operators in C^H. The repeated use of operator `++` means successive increment whereas repeated use of operator `--` indicates successive decrement. These two operators can be combined in any combinations. A single $+$ is treated as an addition or unary plus operator depending on the context. Likewise, a single $-$ can be a subtraction or unary negation operator. For example, following are the valid C^H code.

```
i = +(-9);              /# unary plus and negation operators
i++++;                  /# i = i+2
j = ++i--;              /# i = i+1; j = i; i = i-1;
j = ++++++i;            /# i = 3; j = i;
j = +++++++i--;         /# i = i +3; j = i; i = i - 1;
j = i----;              /# j = i; i = i-2;
i = (*ptr++++)++;       /# ptr = ptr + 2; i = *ptr; *ptr = *ptr + 1;
```

By definition, `++lvalue` means `lvalue = lvalue + 1` and expression `lvalue + 1`, and `lvalue--` is equivalent to expression `lvalue - 1` and `lvalue = lvalue - 1`. The `++` and `--` operators can be applied to any valid lvalues, not just integral variables, so long as the lvalue can add or subtract an integer value of 1 according to internal data conversion rules. Following are the valid C^H code.

```
int i, a[4], *aptr[5];
complex z, *zptr;       /# declare complex variable and complex pointer
z = z++;                /# z = z + 1; z is a complex variable
zptr = (complex *)malloc(sizeof(complex)*90);
aptr[3] = malloc(90);   /# aptr[3] = (int *)malloc(90);
/* imaginary(z)=complex(0.0, 4.0); zptr=zptr+1; *aptr[3]=1; i=i-1 */
imaginary(z) = ++++real(++++++*(zptr+++2*(int)real(++*aptr[3+i--])));
real(z)++;              /# real(z) = real(z) + 1;
----imaginary(*zptr);   /# imaginary(*zptr) = imaginary(*zptr) - 2;
a[--i] = a[2]++;        /# i = i - 1; a[i] = a[2]; a[2] = a[2] + 1;
```

Details about complex numbers and functions **real**() and **imaginary**() in C^H are described in (Cheng, 1993). Note that the memory allocated by function **malloc**() is initialized to zero.

5.8 Type Conversions

In C^H, the explicit type conversion is not necessary in many cases when C needs it as is shown in the previous section for `aptr[3] = malloc(90)`. But, sometimes, it is necessary to convert a value of one type explicitly to a value of other type. This can be achieved by the traditional C cast operation `(type)expr` where `expr` is a C^H expression and `type` is a data type of single object such as **char**, **int**, **float**, **double** or any pointer declaration identifiers such as `char *`, `double *`, `complex *`, etc.. For example, (**int**)9.3, (**float**)ptr, (**double**)9, (**float**$*$)&i, (**complex**$*$)iptr are valid C^H expressions. There is an additional functional type casting operation in C^H in the form of `type(expr)` for data types of single object or `type(expr1, expr2, ...)` for data types of aggregate such as complex and dual (Cheng, 1993; 1993b). In this functional type casting operation, `type` shall not be a pointer data type. For example, **int**(9.3), **complex**(**float**(3), 2), **dual**(2, **float**(3)) are valid C^H expressions. Note that operation **float**() is the same as **real**() if they are used as operands. However, function **real**() can be used as a lvalue as described in (Cheng, 1993) whereas **float**() cannot.

The **sizeof**() function can also use a type identifier. For example, `ptr = malloc(5+sizeof(int*)+sizeof((int)2.3) + sizeof((int)float(90)+7))` is a valid C^H statement.

One important feature of C is its capability for hardware interface by accessing a specific memory location in a computer. This is achieved by pointing a pointer to a specific memory location or register. This hardware interface capability is retained in C^H. For example, the following statements will assign the integer value at the memory location $(68FFE)_{16}$ to variable i and set the byte at the memory address $(FF000)_{16}$ to $(01101001)_2$;

```
char *cptr;
int i, *iptr, j;
iptr = (int *)0X68FFE;     /# iptr points to the memory location at 0X68FFE
i = *iptr;                 /# i equals the value at 0X68FFE;
cptr = (char *)0XFF000;    /# cptr points to the memory location at 0XFF000
*cptr = 0B01101001;        /# 0B01101001 is assigned to 0XFF000
cptr = (float *)cptr + 1;  /# cptr  points to 0XFF004, not 0XFF001.
                           /# note: (float *)cptr++ is (float *)(cptr++)
j = int(cptr);             /# j becomes 0XFF004
```

Note that an integral value cannot be assigned to a pointer variable without an explicit type cast, and vice versa. The lower segment of the memory in a computer is usually reserved for the operating system and system programs. An application program will be terminated with exception handling if these protected segments of memory are messed up by pointers.

6 Real Functions

A computer language with no mathematical functions is not suitable for scientific computing and many other applications. The C language is a small language; it does not provide mathematical functions internally. The mathematical functions are provided in a standard library of mathematical functions. Writing good mathematical functions is not easy as pointed out by Plauger (1992). The mathematical functions implemented in (Plauger, 1992) have provisions for handling of $-$Inf, Inf, and NaN; but they do not distinguish -0.0 from $+0.0$, which is the case for most implementations of mathematical functions in C. Since C does not provide mathematical functions internally, like arithmetic operations in K&R C, the returned value from a standard mathematical function is a double floating-point number regardless of the data types of the input arguments. In some of C implementations, if the input arguments are not doubles the mathematical functions may return erroneous results without warning. Numerically-oriented programmers have little tolerance with respect to the implicit conversion of the data type from float to double for arithmetic operations of a computer language as discussed in section 3. However, they generally accept the strongly typed implementation of mathematical functions. Note that the ANSI C mathematical standard library does not provide any float functions. If a different return data type is desired for a mathematical function, a new function with a different name will be needed. For example, the operation **sin**(1) appears right in C. Indeed, most C programs will execute this operation calmly, but, maybe with an erroneous result because the input data type of integer is not what **sin**() function expected. As another example, the function **abs**() in C returns an absolute int number while **fabs**() will result in a double number. If one wants to get a float absolute value, a new function has to be created. As a result, one has to remember many arcane names for different functions.

The external functions of C^H can be created in the same manner as in C. Unlike C, however, the commonly used mathematical functions are built internally into C^H. The mathematical functions in C^H can handle different data types of the arguments gracefully. The output data type of a function depends on the data types of the input arguments, which is called *polymorphism*. Like arithmetic operators, the built-in commonly used mathematical functions in C^H are polymorphic. For example, for the polymorphic function **abs**(), if the data type of the input argument is int, it will return an int as the absolute value. If the input argument of **abs**() is a float or double, the output will return the same data type of float or double, respectively. For a complex number input, the result of **abs**() is a float with the value of the modulus of the input complex number. Similarly, if the argument data type is lower than or equal to float, **sin**() will return a float result correctly. Function **sin**() can also return double and complex results for double and complex input arguments, respectively. Because I/O functions are also built into C^H itself, different data types are reconciled inside C^H. For example, `printf("%f", x)` in C can print x if x is a float. But, if x is changed to int in a program, the printing statement must also be changed accordingly as `printf("%d", x)`. Therefore, the change of data type declaration of a variable will have to accompany the change of many other parts of the program. Unlike C, the commands **printf**(x) and **printf**(sin(x)) in C^H can handle different data types of x; x can be char, int, float, double, or complex.

For portability, all mathematical functions included in the ANSI C header `math.h` have been implemented polymorphically in C^H. The names of built-in mathematical functions of C^H presented in this paper are based upon the ANSI C header `math.h`. However, one can change, add, or remove these functions and operators in C^H at his/her discretion. These mathematical functions are ANSI C compatible. If the arguments of these functions have the data types of the corresponding ANSI C mathematical functions, there is no difference of these functions between the ANSI C and C^H from a user's point of view. Besides the aforementioned polymorphic nature, the mathematical function in C^H is more powerful due to its abilities to handle metanumbers.

The ANSI C standard is descriptive, many special cases are implementation-dependent. Most mathematical operations related to the metanumbers have not been spelled out in the ANSI C standard. Therefore, loosely speaking, the built-in polymorphic mathematical functions of C^H are ANSI C compatible. Note that, unlike ANSI C, polymorphic mathematical function names in this section, by default, are keywords. In the same token, Inf and NaN are keywords in C^H. The metanumbers Inf and NaN are handled as system constants in the similar manner as constants such as 2.0. Therefore, a declaration statement like

```
int Inf, NaN, sin;
```

is not valid in C^H by default. However, keywords and symbols in C^H can be added, changed, and removed by the built-in functions `addkey(char *old_resword_or_symbol, *new_resword_or_symbol)`, `chkey(char *old_resword_or_symbol, *new_resword_or_symbol)`, and `remkey(char *resword_or_symbol)`, respectively. For example, the command `addkey("fabs","abs")` will make function `fabs()` the same as function **abs**(). The following C^H program is valid.

```
chkey("sin", "SIN");
addkey("printf", "write");
addkey("double", "double_precision");
addkey("=", "equals");
addkey("+", "plus");
begin                          /# real code begins here {
  double_precision sin;        /# double sin;
  sin equals SIN(30) plus 6;   /# sin = SIN(30)+6;
  write("The keyword changeability is unique to CH.\n");
end                            /# end }
```

where `chkey("sin", "SIN")` changes the keyword `sin` to `SIN`. Once a default keyword `sin` is changed, it then can be used as a regular variable. However, using function names in a standard library as variable names is considered a bad programming practice. Whether an object is a keyword can be tested by the built-in function `iskey(char *name)`. The case sensitivity for a C^H program can be switched off and on through a boolean switch function `casesen`(onoff). Therefore, porting code written in other languages and software packages to C^H is not very difficult due to the *keyword changeability*. The detailed exploration of this unique C^H feature is beyond the scope of this paper.

In this section, the built-in mathematical functions of C^H will be discussed. The input and output of the functions involving the metanumbers will be highlighted. The results of the mathematical functions involving metanumbers are given in Tables 13 to 16. In Tables 13 to 16, unless indicated otherwise, x, x_1, x_2 are real numbers with $0 < x, x_1, x_2 < \infty$; and k is an integral value. The value of `pi` is the finite representation of the irrational number π in floating-point numbers. The returned data of a function is float or double depends on the data type of the input arguments. In Table 13, if the order of the data type `x` is less than or equal to float, the returned data type is float. The returned data type is double if `x` is a double data. If the argument `x` of a function in Table 13 is NaN, the function will return NaN. In Tables 14 to 16, the returned data type will be the same as the higher order data type of two input arguments if any of two arguments is float or double. Otherwise, the float is the default returned data type.

Functions defined in this section will return float or double, except for functions **abs**() and **pow**(). If arguments of these two functions are integral values, the returned data types are ints. For example, **pow**(2,16) will return the integral value of 65536. In C^H, if the exponent of the second argument of function **pow**() is an integral value, the computation will be more efficient than its real counterpart. For example, **pow**(x, 3) is more efficient than **pow**(x,3.0). Function **pow**() will optimize the performance for applications that involve a large amount of integer exponentiation. Function **pow**() behaves like the exponentiation operator `**` in Fortran. Note that ANSI C forces function **pow**() to deliver a double data, which not only inhibits the optimization for integer exponentiation, but also changes the data type of an integral expression into a floating-point expression due to the internal data type conversion. This is not applicable for many applications.

The absolute function **abs**(x) will compute the absolute value of an integer or a floating-point number. The absolute value of a negative infinity $-\infty$ is a positive infinity ∞.

The **sqrt**(x) function computes the nonnegative square root of `x`. If `x` is negative, the result is NaN, except that **sqrt**$(-0.0) = -0.0$ according to the IEEE 754 standard. The square root of infinity **sqrt**(∞) is infinity.

The **exp**(x) function computes the exponential function of `x`. The following results hold: $e^{-\infty} = 0.0; e^{\infty} = \infty; e^{\pm 0.0} = 1.0$.

The **log**(x) function computes the natural logarithm of `x`. If `x` is negative, the result is NaN. The value of -0.0 is considered equal to 0.0 in this case. The following results hold: **log**$(\pm 0.0) =$

Table 13: Results of real functions for $\pm 0.0, \pm\infty$, and NaN.

function	x value and results						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
abs(x)	Inf	x_1	0.0	0.0	x_2	Inf	NaN
sqrt(x)	NaN	NaN	-0.0	0.0	sqrt(x)	Inf	NaN
exp(x)	0.0	e^{-x_1}	1.0	1.0	e^{x_2}	Inf	NaN
log(x)	NaN	NaN	$-$Inf	$-$Inf	$\log(x_2)$	Inf	NaN
log10(x)	NaN	NaN	$-$Inf	$-$Inf	$\log_{10}(x_2)$	Inf	NaN
sin(x)	NaN	$-\sin(x_1)$	-0.0	0.0	$\sin(x_2)$	NaN	NaN
cos(x)	NaN	$\cos(x_1)$	1.0	1.0	$\cos(x_2)$	NaN	NaN
tan(x)	NaN	$-\tan(x_1)$	-0.0	0.0	$\tan(x_2)$	NaN	NaN
	Note: $\tan(\pm\pi/2 + 2 * k * \pi) = \pm$Inf						
asin(x)	NaN	$-\mathrm{asin}(x_1)$	-0.0	0.0	$\mathrm{asin}(x_2)$	NaN	NaN
	Note: asin(x) = NaN, for $\lvert x \rvert > 1.0$						
acos(x)	NaN	$\mathrm{acos}(x_1)$	pi/2	pi/2	$\mathrm{acos}(x_2)$	NaN	NaN
	Note: acos(x) = NaN, for $\lvert x \rvert > 1.0$						
atan(x)	$-$pi/2	$-\mathrm{atan}(x_1)$	-0.0	0.0	$\mathrm{atan}(x_2)$	pi/2	NaN
sinh(x)	$-$Inf	$-\sinh(x_1)$	-0.0	0.0	$\sinh(x_2)$	Inf	NaN
cosh(x)	Inf	$\cosh(x_1)$	1.0	1.0	$\cosh(x_2)$	Inf	NaN
tanh(x)	-1.0	$-\tanh(x_1)$	-0.0	0.0	$\tanh(x_2)$	1.0	NaN
asinh(x)	$-$Inf	$-\mathrm{asinh}(x_1)$	-0.0	0.0	$\mathrm{asinh}(x_2)$	Inf	NaN
acosh(x)	NaN	NaN	NaN	NaN	$\mathrm{acosh}(x_2)$	Inf	NaN
	Note: acosh(x) = NaN, for $x < 1.0$; acosh(1.0) = 0.0						
atanh(x)	NaN	$-\mathrm{atanh}(x_1)$	-0.0	0.0	$\mathrm{atanh}(x_2)$	NaN	NaN
	Note: atanh(x) = NaN, for $\lvert x \rvert > 1.0$; atanh($\pm$1.0) = $\pm$Inf						
ceil(x)	$-$Inf	$\mathrm{ceil}(-x_1)$	-0.0	0.0	$\mathrm{ceil}(x_2)$	Inf	NaN
floor(x)	$-$Inf	$\mathrm{floor}(-x_1)$	-0.0	0.0	$\mathrm{floor}(x_2)$	Inf	NaN
ldexp(x, k)	$-$Inf	$\mathrm{ldexp}(-x_1, k)$	-0.0	0.0	$\mathrm{ldexp}(x_2, k)$	Inf	NaN
modf(x, &y)	-0.0	$\mathrm{modf}(-x_1, \&y)$	-0.0	0.0	$\mathrm{modf}(x_2, \&y)$	0.0	NaN
y	$-$Inf	y	-0.0	0.0	y	Inf	NaN
frexp(x, &k)	$-$Inf	$\mathrm{frexp}(-x_1, \&k)$	-0.0	0.0	$\mathrm{frexp}(x_2, \&k)$	Inf	NaN
k	0	k	0	0	k	0	0

Table 14: Results of the function pow(y, x) for $\pm 0.0, \pm\infty$, and NaN.

pow(y, x)											
y value	x value										
	−Inf	−x1	$-2k-1$	−2k	−0.0	0.0	2k	$2k+1$	x2	Inf	NaN
Inf	0.0	0.0	0.0	0.0	NaN	NaN	Inf	Inf	Inf	Inf	NaN
y2 > 1	0.0	$y_2^{-x_1}$	y_2^{-2k-1}	y_2^{-2k}	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	Inf	NaN
1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	NaN	NaN
0 < y2 < 1	Inf	$y_2^{-x_1}$	y_2^{-2k-1}	y_2^{-2k}	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	0.0	NaN
0.0	Inf	Inf	Inf	Inf	NaN	NaN	0.0	0.0	0.0	0.0	NaN
−0.0	Inf	Inf	−Inf	Inf	NaN	NaN	0.0	−0.0	0.0	0.0	NaN
−y1	NaN	NaN	$-y_1^{-2k-1}$	y_1^{-2k}	NaN	NaN	y_1^{2k}	$-y_1^{2k+1}$	NaN	NaN	NaN
−Inf	NaN	NaN	−0.0	0.0	NaN	NaN	Inf	−Inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 15: Results of the function atan2(y, x) for $\pm 0.0, \pm\infty$, and NaN.

atan2(y, x)							
y value	x value						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	3∗pi/4	pi/2	pi/2	pi/2	pi/2	pi/4	NaN
y2	pi	$\text{atan2}(y_2, -x_1)$	pi/2	pi/2	$\text{atan2}(y_2, x_2)$	0.0	NaN
0.0	pi	pi	pi	0.0	0.0	0.0	NaN
−0.0	−pi	−pi	−3∗pi/4	−pi/2	−0.0	−0.0	NaN
−y1	−pi	$\text{atan2}(-y_1, -x_1)$	−pi/2	−pi/2	$\text{atan2}(-y_1, x_2)$	−0.0	NaN
−Inf	−3∗pi/4	−pi/2	−pi/2	−pi/2	−pi/2	−pi/4	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 16: Results of the function fmod(y, x) for $\pm 0.0, \pm\infty$, and NaN.

fmod(y, x)							
y value	x value						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
y2	y_2	$\text{fmod}(y_2, -x_1)$	NaN	NaN	$\text{fmod}(y_2, x_2)$	y_2	NaN
0.0	0.0	0.0	NaN	NaN	0.0	0.0	NaN
−0.0	−0.0	−0.0	NaN	NaN	−0.0	−0.0	NaN
−y1	y_1	$\text{fmod}(-y_1, -x_1)$	NaN	NaN	$\text{fmod}(-y_1, x_2)$	$-y_1$	NaN
−Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

$-\infty$; $\mathbf{log}(\infty) = \infty$. The `log10`(x) function computes the base-ten logarithm of `x`. If `x` is negative, the result is a NaN. Like the function **log**(), the value of -0.0 is considered equal to 0.0. The following results hold: $\mathbf{log10}(\pm 0.0) = -\infty$; $\mathbf{log10}(\infty) = \infty$.

The trigonometric functions **sin**(x), **cos**(x), and **tan**(x) computes sine, cosine, and tangent of `x` measured in radians, respectively. The sine and tangent are odd functions so that $\mathbf{sin}(\pm 0.0) = \pm 0.0$ and $\mathbf{tan}(\pm 0.0) = \pm 0.0$. The cosine is an even function so that $\mathbf{cos}(\pm 0.0) = 1.0$. When the value of the argument is positive or negative infinity, all these functions return NaNs. Theoretically, it is true that $\mathbf{tan}(\pm\pi/2 + 2 * k * \pi) = \pm\infty$. But, in practice, because the irrational number π cannot be represented exactly in float or double data, the **tan**(x) function will never return infinities of $\pm\infty$. Notice that the function **tan**() is not continuous at $\pi/2$, $\mathbf{tan}(\pi/2 - \varepsilon) = \infty$ and $\mathbf{tan}(\pi/2 + \varepsilon) = -\infty$, where ε is a very small number. Due to the finite precision and round-off errors of floating-point numbers, one may get a wrong result near the value of $\pi/2$.

The properties of odd functions of sine and tangent are reflected in their inverse functions **asin**(x) and **atan**(x). The **asin**(x) function computes the principal value of the arc sine of `x`. When the value of `x` is in the range of $[-1.0, 1.0]$, the **asin**(x) function returns the value in the range of $[-\pi/2, \pi/2]$ radians. When `x` is outside the range of $[-1.0, 1.0]$, the arc sine is undefined and **asin**(x) returns NaN. The range of the input value for the even function **acos**(x) of arc cosine is the same as that of **asin**(x). The **acos**(x) function computes the principal value of the arc cosine of `x`. The range of the principal value of the arc cosine is $[0.0, \pi]$ radians. The **atan**(x) function computes the principal value of the arc tangent of `x`. The **atan**(x) function returns the value in the range of $[-\pi/2, \pi/2]$ radians. The following results hold: $\mathbf{atan}(\pm\infty) = \pm\pi/2$.

Like trigonometric functions **sin**(x) and **tan**(x), the hyperbolic functions **sinh**(x) and **tanh**(x) are odd functions. The **sinh**(x) and **tanh**(x) functions compute the hyperbolic sine and tangent of `x`, respectively. The even function **cosh**(x) computes the hyperbolic cosine of `x`. The following results hold: $\mathbf{sinh}(\pm 0.0) = \pm 0.0$; $\mathbf{cosh}(\pm 0.0) = 1.0$; $\mathbf{tanh}(\pm 0.0) = \pm 0.0$; $\mathbf{sinh}(\pm\infty) = \pm\infty$; $\mathbf{cosh}(\pm\infty) = \infty$; $\mathbf{tanh}(\pm\infty) = \pm 1.0$;

The inverse hyperbolic functions are not defined by the ANSI C standard. In $\mathrm{C^H}$, the inverse hyperbolic sine, cosine, and tangent are defined as **asinh**(x), **acosh**(x), and **atanh**(x), respectively. For the **acosh**(x) function, if the argument is less than 1.0, it is undefined and **acosh**(x) returns NaN. **acosh**(1.0) returns a positive zero. The valid domain for function **atanh**(x) is $[-1.0, 1.0]$. The following results hold: $\mathbf{asinh}(\pm 0.0) = \pm 0.0$; $\mathbf{asinh}(\pm\infty) = \pm\infty$; $\mathbf{acosh}(\infty) = \infty$; $\mathbf{atanh}(\pm 0.0) = \pm 0.0$; $\mathbf{atanh}(\pm 1.0) = \pm\infty$.

The **ceil**(x) function computes the smallest integral value not less than the value of `x`. The counterpart of **ceil**(x) is the function **floor**(x) which computes the largest integral value not greater than the value of `x`. The following results hold: $\mathbf{ceil}(\pm 0.0) = \pm 0.0$; $\mathbf{floor}(\pm 0.0) = \pm 0.0$; $\mathbf{ceil}(\pm\infty) = \pm\infty$; $\mathbf{floor}(\pm\infty) = \pm\infty$

The **ldexp**(x, k) function multiplies the value of the floating-point number x with the value of 2 raised to the power of k. The returned value of $x * 2^k$ keeps the sign of x.

The functions **modf**(x, xptr) and **frexp**(x, iptr) have two arguments. The first argument is the input data and the second argument is a pointer which will store the resulted integral part of the function call. The **modf**(x, xptr) function breaks the argument `x` into integral and fractional parts, each of which has the same sign as the argument. The **modf**() function returns the fractional part and the integral part is stored to the memory pointed to by the second argument. The basic data types of two arguments must be the same. For example, if the first argument `x` is float, the second argument `xptr` must be a pointer to float. If the first argument is a metanumber, the integral part will equal the metanumber while the fractional part becomes zero with the sign of the first argument except for NaN. The **frexp**(x, iptr) function breaks a floating-point number into a normalized fraction and an integral power of 2 in the form of $x * 2^k$. The **frexp**(x, iptr) function returns the normalized fraction and the integral part is stored to the memory pointed to by the second argument which is a pointer to int. If the first argument is a metanumber, the fractional part will equal the metanumber while the integral part becomes zero.

The mathematical functions **pow**(y, x), **atan2**(y,x), and **fmod**(y,x) have two input arguments. The results of these three functions are given in Tables 14-16. The **pow**(y, x) function computes `y` raised to the power of `x`, which is y^x or $e^{x\log(y)}$. If x is negative, y^x becomes $1/y^{|x|}$ with the defined division operation given in Table 6. If y is less than zero and x is not an integral value, the function is undefined. The value of -0.0 is considered equal to 0.0 in the evaluation of $\log(-0.0)$ when the value of x is not an integral number. When x is an odd integer number and y is negative, the result is negative. If both y and x are zeroes, 0^0 is indeterminate. For a positive value of y, the result depends on the value of y when x is infinity. If y is less than 1, y^∞ is 0.0; 1.0^∞ is indeterminate; if y is greater than 1, y^∞ is infinity. If y is infinity and x is zero, $(\pm\infty)^{\pm 0.0}$ are indeterminate. It has been suggested that $x^{0.0} = 1$ for any x, including 0.0, Inf, and NaN (Kahan, 1986; Thomas, 1993), which has been implemented in many computer systems including (Sun, 1990a). It is true that if $f(x)$ and $g(x)$ are analytic at a, and $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$; then $\lim_{x\to a} f(x)^{g(x)} = 0^0 = 1$. For example, $\lim_{x\to 0} x^x = 1$ and $\lim_{x\to 0} x^{\sin(x)} = 1$. It is not difficult to find examples that $0^0 \neq 1$ such as in $\lim_{x\to a} x^{\log(x)} = e$ and $\lim_{x\to a}(e^{-1/x})^x = 1/e$. To ensure the proper flow, a $\mathrm{C^H}$ program shall not stop during the execution due to invalid operations. $\mathrm{C^H}$ is designed to be deterministic; all operations and built-in functions either deliver correct numerical results, including Inf, or NaN. It is a bad design for a computer language if at one point it can deliver a correct numerical result while at other point it returns a wrong numerical result. In general, whenever there is a problem in defining the value for a function or operation mathematically, the corresponding $\mathrm{C^H}$ expression will return NaN. Because $\mathrm{C^H}$ expressions such as $1/\mathbf{log}(0.0)$ and $\mathbf{exp}(1/-0.0)$ evaluate to 0.0, therefore, **pow**(0.0,0.0) is defined as NaN in $\mathrm{C^H}$. For the same reason, **pow**(Inf, 0.0) and **pow**(NaN, 0.0) are also defined as NaN. Due to the similar considerations, our decision on x^0 has concurred with the proposed standard for Ada (Dritz, 1991a,b). But, the definition given in Table 14 is much more inclusive than what proposed for Ada. In general, all mathematically indeterminate expressions are defined as NaN in $\mathrm{C^H}$. For an interesting historical debate about whether 0^0 equal 1 or 0^0 is undefined, one can see (Knuth, 1992).

The **atan2**(y, x) function computes the principal value of the arc tangent of `y/x`, using the signs of both arguments to determine the returned value in the range of $[-\pi, \pi]$ radians. Given the (x, y) coordinates of a point in the X-Y plane, the **atan2**(y, x) function computes the angle of the radius from the origin to the point. Any positive number which overflows is represented by Inf. The negative overflow is $-$Inf. The following results hold: $\mathbf{atan2}(\pm\mathrm{Inf}, -\mathrm{Inf}) = \pm 3\pi/4$; $\mathbf{atan2}(\pm\mathrm{Inf}, \mathrm{Inf}) = \pm\pi/4$; $\mathbf{atan2}(\pm\mathrm{Inf}, \mathrm{x}) = \pm\pi/2$; $\mathbf{atan2}(\pm y, \mathrm{Inf}) = \pm 0.0$; and $\mathbf{atan2}(\pm y, -\mathrm{Inf}) = \pm\pi$. When both values of y and x are zeroes, the function **atan2**(y, x) will return the results consistent with the manipulation of metanumbers discussed so far. The value of -0.0 is considered as a negative number less than zero. Therefore, the following results are defined for these special operations: $\mathbf{atan2}(0.0, -0.0) = \pi$; $\mathbf{atan2}(0.0, 0.0) = 0.0$; $\mathbf{atan2}(-0.0, -0.0) = -3\pi/4$; and $\mathbf{atan2}(-0.0, 0) = -\pi/2$, which is consistent with the treatment of the metanumbers of $\pm$Inf in $\mathbf{atan2}(-\mathrm{Inf},\ -\mathrm{Inf}) = -3pi/4$. In $\mathrm{C^H}$, $\mathbf{atan2}(0.0, 0.0)$ is a specially defined value. These results are different from those by the SUN's ANSI C compiler, which is in conformance with 4.3 Berkeley Software Delivery (SUN, 1990a). According to 4.3BSD, the results for these special cases are $\mathbf{atan2}(\pm 0.0, -0.0) = \pm 0.0$ and $\mathbf{atan2}(\pm 0.0, 0.0) = \pm\pi$, which implies that the values of ± 0.0 in x-axis are different from those in y-axis.

The **fmod**(y,x) function computes the floating-point remainder of `y/x`. The **fmod**(y,x) function returns the value of $y - i * x$ for some integer i. The magnitude of the returned value with the same sign of x is less than the magnitude of x. If x is zero, the function is undefined and returns NaN.

When y is infinity, the result is also undefined. If x is infinity and y is a finite number, the result is the same as y.

7 Programming Examples

7.1 Computation of Extreme Values of Floating-Point Numbers

Due to different machine architectures for representation of floating-point numbers, the extreme values such as the maximum representable floating-point value are different. For two machines with the same representation of floating-point values, the same operations such as adding two values on each machine may get different results, depending on the schemes for rounding a number which cannot be represented exactly. To aid serious numerically-oriented programmers in writing their programs, the ANSI C standard added header `float.h` as a companion to the existing header `limits.h` which deals with the machine-dependent integer values only. In this section, we will show how parameters defined in the ANSI C standard library `float.h` can be computed in C^H without knowing the intricate architecture of the computer. A program can less depend on these parameters if a language can support metanumbers Inf and NaN. The use of metanumbers such as Inf and NaN instead of parameters is recommended for C^H programming.

7.1.1 Minimum Floating-Point Numbers FLT_MIN and FLT_MINIMUM

The parameter FLT_MIN is defined in the ANSI C standard library header `float.h` as a minimum normalized positive floating-point float number. If a number is less than FLT_MIN, it is called an *underflow*. Since the IEEE 754 standard provides a *gradual underflow*, the minimum denormalized positive floating-point float number is defined as FLT_MINIMUM in C^H. Because of gradual underflow, the C^H expression `x - y == 0` is TRUE iff x = y, which is not true for systems that lack gradual underflow. This parameter is very useful from a programming point of view. As an example, assume that values of FLT_MINIMUM and FLT_MIN are 1.401298e-45 and 1.175494e-38, respectively. The following C^H code will illustrate subtleties of these two parameters.

```
float f, *flt_minimum;
int minimum, i;
minimum = 1;                       /# memory location becomes 00000001
flt_minimum = &minimum;            /# *flt_minimum becomes FLT_MINIMUM
i = *flt_minimum > 0.0;            /# i becomes 1
i = FLT_MIN > *flt_minimum;        /# i becomes 1
i = fabs(*flt_minimum) > 0.0;      /# i becomes 1
f = (*flt_minimum)/(*flt_minimum); /# f becomes 1.0; note 0.0/0.0 = NaN
f = f/1.e-46                       /# f becomes Inf: note 1.e-46 < FLT_MINIMUM
```

Applications of these two numbers in handling of branch cuts of multiple-valued complex functions will be described in (Cheng, 1993).

7.1.2 Machine Epsilon FLT_EPSILON

The machine epsilon `FLT_EPSILON` is the difference between 1 and the least value greater than 1 that is representable in float. This parameter, defined in the ANSI C header `float.h`, is a system constant in C^H. This parameter is very useful for scientific computing. For example, due to the finite precision of the floating-point representation and alignment of addition operation, when a significantly small value and a large number are added together, the small number may not have contribution to the summation. Using FLT_EPSILON, whether adding a small positive number x to a large positive number y can at least capture three decimal digits of significance of y can be tested by

```
if(x < y * FLT_EPSILON * 1000)
```

The following C^H code can calculate and print out the machine epsilon on the screen

```
float epsilon;
epsilon = 1.0;
while(epsilon+1 > 1)
  epsilon /= 2;
epsilon *= 2;
printf("The machine epsilon FLT_EPSILON is %e", epsilon);
```

For SUN SPARCStations, the output from the execution of the above code is as follows:
The machine epsilon FLT_EPSILON is 1.192093e-07
which matches the value of the parameter `FLT_EPSILON` defined in the ANSI C header `float.h`. Although the above computation of the parameter `FLT_EPSILON` is simple in C^H which uses the default rounding mode of round toward nearest, it may be vulnerable to other rounding modes. A more robust method (Plauger, 1992) to obtain this parameter is by manipulating the bit pattern of the memory of a float variable as shown in section 7.1.1.

7.1.3 Maximum Floating-Point Number FLT_MAX

The parameter `FLT_MAX` defined in the ANSI C header `float.h` is the maximum representable finite floating-point number. Any value that is larger than `FLT_MAX` will be represented as Inf and any value less than −`FLT_MAX` is represented by −Inf. If the value of `FLT_MAX` is represented as $fltmax * 10^e$, then the following two equations will be satisfied

$$(fltmax + FLT\_EPSILON) * 10^e = Inf$$

$$(fltmax + FLT\_EPSILON/2) * 10^e = FLT\_MAX$$

where the machine epsilon FLT_EPSILON is defined in the previous section 7.1.2 and exponential value e is to be calculated. The following C^H program will calculate FLT_MAX as well as FLT_MAX_10_EXP and FLT_MAX_EXP of the machine and print them on the screen. The value of `FLT_MAX_10_EXP` is the maximum integer such that 10 raised to its power is in the range of the representable finite floating-point numbers. The value of `FLT_MAX_EXP` is the maximum integer such that 2 raised to its power minus 1 is a representable finite floating-point number. For the illustrative purpose, only the while-loop control structure is used in this example.

```
float b, f, flt_max;
int e, i, flt_max_exp, flt_max_10_exp;
b = 10; e = 0; f = b;
/* calculate exponential number e, 38 in the example */
while(f != Inf)
{
  e++; f*=b;
```

```
}
flt_max_10_exp = e;
/* calculate leading non-zero number, 3 in the example */
i = 0; f = 0.0;
while(f != Inf)
  f = ++i * pow(b, e);
/* calculate numbers after decimal point, 40282347... in the example */
flt_max = i;
while(e != 0)
{
  flt_max = --flt_max * b;
  e--; i = 0; f = 0.0;
  while( f != Inf && i < 10)
  {
    f = ++flt_max * pow(b, e);
    i++;
  }
}
f = frexp(flt_max, &flt_max_exp);       /# calculate FLT_MAX_EXP
printf("FLT_MAX = %.8e \n", flt_max);
printf("FLT_MAX (in binary format) = %b \n", flt_max);
printf("FLT_MAX_10_EXP = %d \n", flt_max_10_exp);
printf("FLT_MAX_EXP = %d \n", flt_max_exp);
```

The output of the above code on SUN SPARCStations is as follows:
FLT_MAX = 3.40282347e+38
FLT_MAX (in binary format) = 01111111011111111111111111111111
FLT_MAX_10_EXP = 38
FLT_MAX_EXP = 128
The above values for `FLT_MAX`, `FLT_MAX_10_EXP`, and `FLT_MAX_EXP` are the same as the parameters defined in the ANSI C header `float.h`. By just changing the declaration of the first statement from float to double, the corresponding extreme values `DBL_MAX`, `DBL_MAX_10_EXP`, and `DBL_MAX_EXP` for double can be obtained. In this case, the polymorphic arithmetic operators and mathematical functions **pow**() and **frexp**() will return double data. Note that the default mode for floating-point constants is float which can be switched to double by function **floatconst**(FALSE).

In the above calculation of the extreme floating-point values, the user does not need to know the intricate machine representation of floating-point numbers. If one knows the machine representation of a floating-point number, the calculation of the extreme values can be much simpler. For example, according to Table 1, the value of `FLT_MAX` is represented in a hexadecimal form as $(7F7FFFFF)_{16}$. The following C^H program can be used to calculate the maximum representable finite floating-point number `FLT_MAX`.

```
int i; float *flt_max;
flt_max = &i;       /# flt_max points to the memory location of i
i = 0X7F7FFFFF;     /# *flt_max becomes FLT_MAX
```

The maximum float number `FLT_MAX` can also be readily obtained by the I/O function **scanf**() with the binary input format `"%32b"`. For interested readers, can you think of any other method

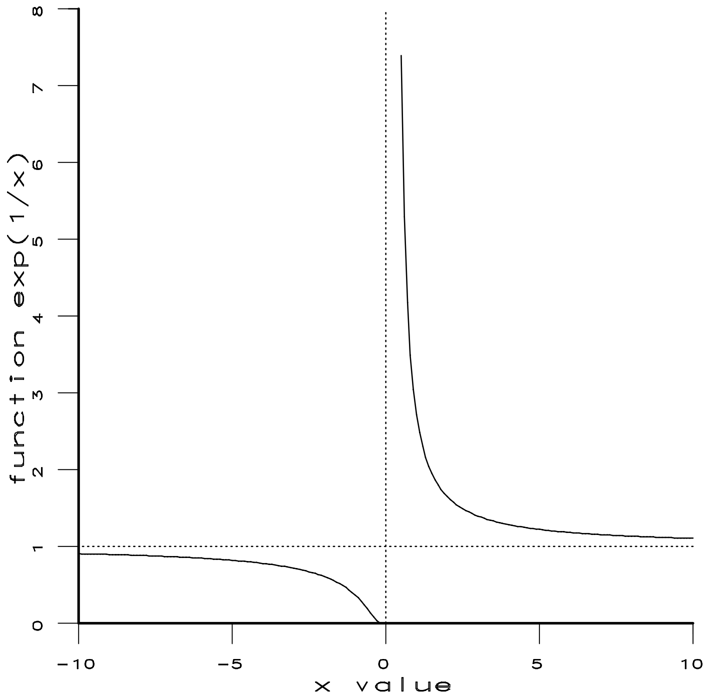

Figure 1: Function $f(x) = e^{\frac{1}{x}}$.

for computing the maximum representable finite floating-point number `FLT_MAX` by a C or Fortran program without knowing the machine architecture? The major difficulty is that, due to the internal alignment for calculation of the floating-point numbers, the significantly small number will be ignored when it is added to or subtracted from a large number. For example, the execution of the command `f = FLT_MAX + 3.0e30` will give the variable `f` the value of FLT_MAX although the value of $3.0 * 10^{30}$ is not a small number, but it is significantly smaller than `FLT_MAX` and ignored in the above addition operation. The following two $\mathrm{C^H}$ expressions will further demonstrate the difference between FLT_MAX and Inf, 1/Inf∗FLT_MAX = 0.0 and 1/FLT_MAX∗FLT_MAX = 1.0.

7.2 Programming with Metanumbers

The $\mathrm{C^H}$ language distinguishes −0.0 from 0.0 for real numbers. The metanumbers 0.0, −0.0, Inf, −Inf, and NaN are very useful for scientific computing. For example, the function $f(x) = e^{\frac{1}{x}}$ is not continuous at the origin as is shown in Figure 1. This discontinuity can be handled gracefully in $\mathrm{C^H}$. The evaluation of the $\mathrm{C^H}$ expression **exp**(1/0.0) will return Inf and **exp**(1/(−0.0)) gives 0.0, which corresponds to mathematical expressions $e^{\frac{1}{0+}}$ and $e^{\frac{1}{0-}}$ or $\lim_{x\to 0_+} e^{\frac{1}{x}}$ and $\lim_{x\to 0_-} e^{\frac{1}{x}}$, respectively. In addition, the evaluation of expressions **exp**(1.0/Inf) and **exp**(1.0/(−Inf)) will get the value of 1.0. As another example, the function `finite(x)` recommended by the IEEE 754 standard is equivalent to the $\mathrm{C^H}$ expression `-Inf < x && x < Inf`, where x can be a float/double variable or expression. If x is a float, `-Inf < x && x < Inf` is equivalent to `-FLT_MAX <= x && x <= FLT_MAX`; If x is a double, `-Inf < x && x < Inf` is equivalent to `-DBL_MAX <= x && x <= DBL_MAX`; The mathematical statement "$if\ -\infty < value <= \infty, then\ y\ becomes\ \infty$" can be easily programmed in $\mathrm{C^H}$ as follows

```
if(-Inf < value && value <= Inf) y = Inf;
```

However, a computer can only evaluate an expression step by step. Although the metanumbers are limits of the floating-point numbers, they cannot replace mathematical analysis. For example, the natural number e equal to 2.718281828... is defined as the limit value of the expression

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = e.$$

But, the value of the expression **pow**(1.0 + 1.0/Inf, Inf) in C$^{\mathrm{H}}$ is NaN. The evaluation of this expression is carried out as follows:

$$\left(1.0 + \frac{1.0}{Inf}\right)^{Inf} = (1.0 + 0.0)^{Inf} = 1.0^{Inf} = NaN$$

If the value FLT_MAX instead of Inf is used in the above expression, the result is obtained by

$$\left(1.0 + \frac{1.0}{FLT\_MAX}\right)^{FLT\_MAX} = (1.0 + 0.0)^{FLT\_MAX} = 1.0^{FLT\_MAX} = 1.0$$

According to rules for negation, subtraction, and equal comparison operations given in Tables 2, 4, and 9, the C$^{\mathrm{H}}$ expression `x-y == -(y-x)` will always return TRUE for any values of x and y with x equal to y, including NaN, ±0.0, and ±Inf. The outcome of this computation really matches our intuition regarding algebra. However, there is a subtle difference between two expressions $x - y$ and $-(x - y)$ in C$^{\mathrm{H}}$. When $x = y$ and NaN $\neq x \neq$ *Inf*, $x - y$ will produce 0.0 whereas $-(x - y)$ will return -0.0. If the IEEE 754 standard for handling NaN in relational operations was strictly followed, the implication of the above operation would be much more complicated.

The application of NaN can be further demonstrated by numerically solving quadratic equation

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

The execution of the following C$^{\mathrm{H}}$ program

```
float root[2];
float a,b,c;
a = 1; b = 2; c = 2;
root[0] = (-b+sqrt(b*b-4*a*c))/(2*a);
root[1] = (-b-sqrt(b*b-4*a*c))/(2*a);
if(root[0] == NaN)
  printf("Solutions are complex numbers.\n");
```

will produce the following output
Solutions are complex numbers.
because solutions to the equation of $x^2 + 2x + 2 = 0$ are $-1 \pm i$. In (Cheng, 1993), this equation will be solved in complex numbers.

Because metanumber NaN is unordered, a program involving relational operations should be handled cautiously. For example, the expression `x > y` is not equivalent to `!(x <= y)` if either x or y is a NaN. As an another example, the following C$^{\mathrm{H}}$ code fragment

```
if(x > 0.0)  function1();
else function2();
```

is different from the code fragment

```
if(x <= 0.0) function2();
else function1();
```

The second `if`-statement should be written as `if(x <= 0.0 || x == NaN)` in order to have the same functionality for these two code fragments.

8 Conclusions

C^H not only retains most features of C from the scientific computing point of view, but also extends C's numerical computational capabilities. Metanumbers of -0.0, 0.0, Inf, $-$Inf, and NaN introduced in C^H are external, which makes the power of the IEEE 754 arithmetic standard easily available to the programmer. Furthermore, These metanumbers are extended to commonly used mathematical functions in the spirit of the IEEE 754 standard. The rules for manipulation of these metanumbers in I/O; arithmetic, relational, and logic operations; and commonly used mathematical functions in C^H are defined in this paper. The C^H extensions related to bitwise, assignment, address and indirection, increment and decrement, as well as type conversion operations to ANSI C have been highlighted. The gradual underflow feature of the IEEE 754 standard has been explored through parameter FLT_MINIMUM. Since the ANSI C standard is descriptive, the rigorous definitions defined in this paper will not violate the standard. Like arithmetic operators, the built-in mathematical functions in C^H are polymorphic, which means that the returned data type of a function depends on the data types of the input arguments. This will simplify the scientific programming significantly.

All points delineated in this paper have been implemented and tested in C^H. Example programs with metanumbers and polymorphic mathematical functions are given in the paper. The function names can be added, removed, and changed; and the mathematical operators can be added and removed in C^H. Therefore, porting code from other languages to C^H is relatively simple. Most C programs can be executed in the C^H environment with a minimum modification related to the interpretive nature of the current implementation of C^H. The extension of scientific programming with real numbers to scientific programming with complex numbers will be addressed in (Cheng, 1993).

9 References

1. ANSI, *ANSI Standard X3.9-1978, Programming Language Fortran*, (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.
2. ANSI, *ANSI/IEEE Standard 770 X3.97-1983, IEEE Standard Pascal Programming Language*, IEEE, Inc., NJ, 1983.
3. ANSI, *ANSI Standard X3.159-1989, Programming Language C*, ANSI, Inc., NY, 1989.
4. Apple Computer, Inc., *Apple Numerics Manual*, Addison-Wesley, 1986.
5. Cheng, H. H., Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP, *(IEEE) Computer*, Vol. 22, No. 9, September, 1989, pp. 31-46.
6. Cheng, H. H., Handling of Complex Numbers in the C^H Programming Language, Department of Mechanical, Aeronautical and Materials Engineering, University of California, Davis, Technical Report TR-MAME-93-102, February 18, 1993; also *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 76-106.

7. Cheng, H. H., Computations of Dual Numbers in the Extended Finite Dual Plane, *Proc. of 1993 ASME Design Automation Conference*, Albuquerque, NM, Sept. 19-22, 1993b, vol. 2, pp. 73-80.
8. Cody, W. J., et al, A Proposed Radix- and Word-Length-Independent Standard for Floating-point Arithmetic, *IEEE Micro*, Vol. 4, No. 4, Aug, 1984, pp. 86-100.
9. Dritz, K. W., Proposed Standard for a Generic Package of Elementary Functions for Ada, *Ada Numerics Standardization and Testing*, *ACM Ada Letters*, Vol. XI, No. 7, Fall, 1991a, pp. 9-46.
10. Dritz, K. W., Rational for the Proposed Standard for a Generic Package of Elementary Functions for Ada, see Dritz 1991a, 1991b, pp. 47-65.
11. Dritz, K. W., Proposed Standard for a Generic Package of Primitive Functions for Ada, see Dritz 1991a, 1991c, pp. 66-82.
12. Dritz, K. W., Rational for the Proposed Standard for a Generic Package of Primitive Functions for Ada, see Dritz 1991a, 1991d, pp. 83-90.
13. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.
14. Hodgson, G. S., Proposed Standard for Packages of Real and Complex Type Declarations and Basic Operations for Ada (including Vector and Matrix Types), see Dritz 1991a, 1991a, pp. 91-130.
15. Hodgson, G. S., Rationale for Proposed Standard for Packages of Real and Complex Type Declarations and Basic Operations for Ada (including Vector and Matrix Types), see Dritz 1991a, 1991b, pp. 131-139.
16. Kahan, W., Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit, *The State of the Art in Numerical Analysis* (ed. A. Iserles & M. J. D. Powell), 1987, Clarendon Press, Oxford; *Proc. of the Joint IMA/SIAM Conference*, Univ. of Birmingham, April 14-18, 1986.
17. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, first edition (K&R C), 1978; second edition (ANSI C), 1988.
18. Knuth, D. E., Two Notes on Notation, *The American Mathematical Monthly*, Vol 99, No. 5, May, 1992, pp.403-422.
19. MathWorks, *Pro-MATLAB User's Guide*, The MathWorks, Inc., 1990.
20. Motorola, Inc., *M68000 Family Programmer's Reference Manual*, 1989a.
21. Motorola, Inc., *MC68881/882 Floating-Point Coprocessor User's Manual*, second ed., 1989b.
22. Plauger, P. J., *The Standard C Library*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.
23. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1990.
24. Ritchie, D. M., Johnson S. C., Lesk, M. E., and Kernighan, B. W., The C Programming Language, *The Bell System Technical Journal*, vol. 57, No. 6, July-August 1978, pp. 1991-2020.
25. Ritchie, D. M. and Thompson, K. L., The Unix Time-Sharing System, *Commun. ACM*, vol. 17, No. 7, July 1974, pp. 365-375.
26. Rosler, L., The Evolution of C — Past and Future, *The Bell System Technical Journal*, vol. 63, No. 8, October 1984, pp. 1685-1699.
27. Stroustrup, B., *The C^{++} Programming Language*, Addison-Wesley, Publishing Company, Inc., 1987.

28. SUN, Mathematical Library, *SunOS Reference Manual*, Vol. II, SUN Microsystems, Inc., 1990a, pp. 1301-1327.
29. SUN, Sun C Data Representation, *Programmer's Language Guides: C Programmer's Guide*, SUN Microsystems, Inc., 1990b, pp. 77-89.
30. Thompson, K., Unix Implementation, *The Bell System Technical Journal*, vol. 57, No. 6, July-August 1978, pp. 1931-1946.
31. Thomas, J., Floating-Point C Extensions, NCEG X3J11.1/93-001, January 20, 1993.
32. Wolfram, S., *Mathematica: a System for Doing Mathematics by Computer*, Addison-Wesley Pub. Co., 1988.