Trans-Floating-Point Arithmetic Removes Nine Quadrillion Redundancies From 64-bit IEEE 754 Floating-Point Arithmetic

James A.D.W. Anderson

***Abstract*—IEEE 754 floating-point arithmetic is widely used in modern, general-purpose computers. It is based on real arithmetic and is made total by adding both a positive and a negative infinity, a negative zero, and many Not-a-Number (NaN) states. Transreal arithmetic is total. It also has a positive and a negative infinity but no negative zero, and it has a single, unordered number, nullity. Modifying the IEEE arithmetic so that it uses transreal arithmetic has a number of advantages. It removes one redundant binade from IEEE floating-point objects, doubling the numerical precision of the arithmetic. It removes eight redundant, relational, floating-point operations and removes the redundant total order operation. It replaces the non-reflexive, floating-point, equality operator with a reflexive equality operator and it indicates that some of the exceptions may be removed as redundant – subject to issues of backward compatibility and transient future compatibility as programmers migrate to the transreal paradigm.**

***Index Terms*—transreal arithmetic, transreal numbers, floating-point arithmetic.**

I. Introduction

IEEE floating-point arithmetic [1] [2] is widely used in modern, general purpose computers. This is both a strength, promoting interoperability of computer programs on different hardware, and a weakness where the standard itself has infelicities. The standard has been revised but its constant feature is that it provides a total computing system in which any floating-point operation can be applied to any floating-point objects, with the result being a floating-point object. But, strictly speaking, it is not an *arithmetic* because it applies to Not-a-Number objects, NaNs, that are not numbers.

The original IEEE standard [1] developed floating-point arithmetic in terms of bit patterns that support a finite model of real arithmetic. But division by zero is not defined in real arithmetic so, to make the floating-point model total, both a positive infinity and a negative infinity were added, along with a negative zero and many NaNs.

Here we develop a suggestion [4] to use transreal arithmetic [5] as the basis for floating-point arithmetic. The next section summarises relevant features of IEEE 754 floating-point arithmetic. We then develop a trans-floating-point arithmetic. Next we compare the two floating-point systems and suggest that a single conceptual failure explains all of the infelicities in the design of the IEEE 754 system. Finally we conclude with a summary of the main original contributions of the paper.

Manuscript received June 1, 2014; revised June 9, 2014.

J.A.D.W. Anderson is with the School of Systems Engineering, Reading University, England, RG6 6AY e-mail: j.anderson@reading.ac.uk

II. IEEE 754 Floating-Point Arithmetic

The IEEE floating-point arithmetic standard is quite complex, running to 58 pages [2]. We summarise only the parts that are immediately relevant to a comparison with trans-floating-point arithmetic. A great deal more could be said in a longer paper.

A. Redundancy

IEEE floating-point arithmetic is defined [1] in terms of floating-point operations on bit patterns, interchange formats, that are stored in memory to represent floating-point objects, and by a set of condition flags or exceptions that handle exceptional cases. An operation may be carried out, in the processor, at a higher precision than the floating-point object's storage class. There are two kinds of storage class: unextended and extended. The revised standard [2] also defines an extendable binary format and a decimal format but we discuss only the binary formats here because they have fewer redundancies. Thus the binary IEEE 754 formats provide stronger comparisons with trans-floating-point numbers, which are irredundant.

An IEEE 754 floating-point object has three parts: a sign bit, an exponent, and a significand (or mantissa). In interchange format, the exponent is an unsigned integer from which a bias is subtracted to provide positive, zero and negative exponents during a floating-point calculation in the processor. Two exponents are reserved: the smallest exponent indicates subnormal numbers and the largest indicates infinities and NaNs. There are two kinds of NaNs, silent NaNs that propagate and signalling NaNs that trigger an exception which may or may not terminate execution depending on how the end-user programmer implements a program. Zero and negative zero have the smallest exponent and all significand bits zero. The zeros are distinguished by the sign bit. The infinities have the highest exponent and all significand bits zero: positive and negative infinity are distinguished by the sign bit. The NaNs also have the highest exponent but the significand has at least one non-zero bit. The NaNs are not distinguished by the sign bit. Taking m as the number of significand or mantissa bits, the number of bit patterns reserved for NaNs is given in [4] as $2^{m+1} - 2$. The $+1$ term arises from the sign bit and the -2 term from the two codes reserved for the signed infinities; of these NaN states, only $2^m - 1$ are distinguished so a total of $2^m - 1$ states are redundant. With a 64-bit storage class, $m = 52$ and the redundancy is approximately 4.5×10^{15} states, in words, four and a half quadrillion states are redundant! The eventual redundancy is twice this number.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

B. Relational Operators

IEEE 754 floating-point arithmetic has both a bitwise relational operator and many floating-point relational operators.

The bitwise operator is implemented as the predicate $\mathrm{totalOrder}(x, y)$ which implements a total order, $x \preceq y$, on floating-point objects in canonical form. The unextended, binary formats enforce canonical form so they are totally ordered. The extended and extendable binary formats may or may not enforce a canonical form so they may or may not be totally ordered. The decimal formats do not enforce a canonical form so they are not totally ordered. Here we write binary equality as $x \simeq y$. Binary equality holds, i.e. $x \simeq y$, when x and y are in canonical form and $\mathrm{totalOrder}(x, y)$ is true and $\mathrm{totalOrder}(y, x)$ is true.

The floating-point relational operators are constructed from the four basic relations: *less than* $(<)$, *equal to* $(=)$, *greater than* $(>)$, *unordered* $(?)$ which are said to be mutually exclusive. Of the possible $2 \times 2^4 = 32$ operators, including negations, only 22 are defined explicitly. The remaining 10 operators are not useful and are not all functionally distinct. For example, the standard [1] [2] states that *not equal to* is identical to *unordered or greater than or less than*. But any lack of distinctness implies that the four, basic, relations are not mutually exclusive, contradicting the claim made in the standard.

The floating-point equality operator, $=$, is not reflexive. Firstly the unequal bit patterns for negative and positive zero compare equal. Secondly all NaN bit patterns compare unequal, even if the bit patterns are identical. The bitwise equality, $x \simeq y$ is reflexive when x, y are in canonical form but is not reflexive otherwise. Thus it is guaranteed to be reflexive only in the unextended, binary formats.

If a floating-point relational operator, without ?, is applied to any NaN object then the result is a signalling NaN which, depending on how the end-user program handles it, will or will not terminate execution.

C. Arithmetic

IEEE 754 'arithmetic' defines a finite model of real arithmetic, augmented with an arithmetic of positive and negative infinity, which is consistent with the handling of infinite limits in mathematics (measure theory and extended real-analysis). It also defines operations on the many NaN objects. Let the binary operator, $\circ$, stand for an arbitrary one of the IEEE 754 binary operations of addition, subtraction, multiplication and division; let f be a non-NaN floating-point object and assume that all floating-point arguments are in the same canonical format then $\mathrm{NaN}_i \circ f \simeq \mathrm{NaN}_i \simeq f \circ \mathrm{NaN}_i$ so the result of operating on exactly one NaN is that NaN but $(\mathrm{NaN}_i \circ \mathrm{NaN}_j \simeq \mathrm{NaN}_i)$ or $(\mathrm{NaN}_i \circ \mathrm{NaN}_j \simeq \mathrm{NaN}_j)$ so that the result of operating on two NaNs is some, unspecified, one of them.

D. Exceptions

IEEE 754 'arithmetic' has both control-flow exceptions, such as an *invalid operation* or other error, and behavioural exceptions, i.e. corner cases. According to the IEEE 754 standard [1] [2], there are many invalid operations and other exceptional states. There are also many corner cases. We give one corner case here: the function $\mathrm{negate}(f)$ is not the same as $\mathrm{subtraction}(0, f)$ so $-f$ is not always identical to $0 - f$.

III. Trans-Floating-Point Arithmetic

The format of trans-floating-point numbers is given in [4] by specifying modifications to the unextended, binary formats for IEEE 754 floating-point arithmetic. Specifically nullity, $\Phi = 0/0$, replaces negative zero so that it is encoded with non-zero sign bit and all other bits zero. The signed transreal infinities, $-\infty = -1/0$ and $\infty = 1/0$, are distinguished by the sign bit and have all other bits non-zero, that is they have the largest representable exponent and magnitude. This format is irredundant so every bit pattern encodes a unique transreal number. This lack of redundancy almost doubles the range of real numbers encoded, falling short by a single unit in each of the positive and negative ranges. Incrementing the exponent's bias, by unity, keeps the real range almost the same, falling short by one unit in each of the positive and negative ranges, but exactly halves the magnitude of the smallest, representable, non-zero number. Compared to IEEE 754 floating-point arithmetic, this trades a total of two units of range for a doubling in precision (it being understood that precision is measured here in terms of the magnitude of the least, non-zero, representable number).

Numerical algorithms usually terminate on a tolerance which is a modest function of the least, representable, positive number. Trans-floating-point arithmetic halves the size of this number, compared to IEEE 754 floating-point arithmetic, so implementations have the possibility of proceeding to twice the accuracy in the same number of bits.

One must take care with negation to avoid the problem that IEEE 754 floating-point arithmetic has where $-f \neq 0 - f$. Negation shall toggle the sign bit of a trans-floating-point number if and only if at least one other bit is non-zero. Hence $\mathrm{negate}(\Phi) = \Phi$ and $\mathrm{negate}(0) = 0$ by identity (the sign bit is not toggled) and all other numbers have $\mathrm{negate}(f) = -f$ where the signs of f and $-f$ are made opposite by toggling. This correctly negates all transreal numbers and is very cheap to implement in hardware, with very fast execution.

Transreal arithmetic [5] is then implemented in the processor and, subject to the IEEE 754 floating-point prescriptions on taking the result to specified units in the last place, the processor should implement the arithmetic at a higher precision and may do so in a redundant format, converting the result to an irredundant, transfer format when the result is transferred to memory or to an output device.

The earlier proposal [4] does not give much consideration to rounding modes and the flagging of exceptions. Space prevents us from doing that here but we could do so in a longer paper. In essence one should keep all of the rounding modes, should add the exception *underflow from negative*, but may dispense with some of the current exceptions. The semantics of transreal arithmetic lead to a clear indication of which exceptions to dispense with, for example *division by zero* is never exceptional and *invalid operation* never occurs so they may be removed as being entirely redundant. Nonetheless they may be desired for backward compatibility or transient compatibility as programmers move from the IEEE 754 paradigm to the transreal paradigm. These issues call for fine judgements of human psychology, commercial priorities and scientific correctness. It would take considerable space to give them proper consideration and would, almost certainly, require consensus building, in various communities, before a *de jure* standard should be established.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

IV. Discussion

Real arithmetic is partial – it fails on division by zero. There are two ways to address the problem of partial performance in any system: one can develop a total system or one can seek to correct each of the, generally, infinitely many consequences of partial performance. Transreal arithmetic and trans-floating-point arithmetic take the former approach and, the evidence examined here suggests, the IEEE 754 standard takes the latter approach. We now compare the two floating-point arithmetics and illustrate how the *design goal of totality* explains their differences in performance.

Trans-real arithmetic was designed to be a total arithmetic which is consistent with real arithmetic and with infinite limits as used in measure theory and (extended) real analysis. It adds three definite numbers to the reals: negative infinity ($-\infty = -1/0$), positive infinity ($\infty = 1/0$) and nullity ($\Phi = 0/0$). There is a machine proof that transreal arithmetic is consistent [5]. By contrast IEEE 754 arithmetic is totalised in an *ad hoc* way, which leads it into a number of difficulties, some of which are discussed here. Nonetheless the two arithmetics agree on all real and infinite calculations, except those infinite calculations that involve IEEE 754's negative zero. Transreal arithmetic does not have a negative zero so it has no expression corresponding to $1/(-0) = -\infty$ and where IEEE 754 has $1/(-\infty) = -0$, transreal arithmetic has $1/(-\infty) = 0$. The two arithmetics disagree in every calculation that involves transreal nullity and IEEE 754 NaNs. For example transreal arithmetic has $\infty - \infty = \Phi$ but IEEE 754 has $\infty - \infty \to \mathrm{NaN}_i \neq \mathrm{NaN}_i$. To be clear, subtracting two transreal infinities produces the unique number nullity as a result, which, like all transreal numbers, is equal to itself. This justifies writing the equals sign ($=$) in $\infty - \infty = \Phi$. By contrast, subtracting two IEEE 754 infinities produces some unspecified NaN, which justifies both the production rule arrow ($\to$) and the index (i) on the NaN, but no NaN is equal to itself, which justifies the not-equals sign ($\neq$) in $\infty - \infty \to \mathrm{NaN}_i \neq \mathrm{NaN}_i$. In short, transreal arithmetic has $\infty - \infty = \Phi$ where IEEE 754 has $\infty - \infty \neq \mathrm{NaN}_i$ for all i.

As an abstract mathematical system, transreal arithmetic does not need a negative zero, nor do finite computer models of transreal arithmetic. It is sufficient to switch execution paths on an underflow from a negative number to zero so that any division by that zero operates on a negated numerator. Suppose we want to compute $k/0$ for some positive k. In the case that the zero is exact or produced by an underflow from a positive number, both arithmetics compute $k/0 = \infty$. In the case that the zero is produced by underflow from a negative number, IEEE 754 floating-point arithmetic computes $k/(-0) = -\infty$ in a single execution path and trans-floating-point arithmetic switches to a second execution path to compute $-k/0 = -\infty$. Thus both arithmetics compute an equivalent result. The switch on *underflow from negative* is an additional cost that trans-floating-point arithmetic pays, conditionally, when dividing by zero. This is a consequence of taking a finite approximation to transreal arithmetic. The benefit is that it gains a simpler semantics than IEEE 754 floating-point arithmetic. This is an issue which would take a great deal of space to explore in a longer paper.

There are three basic, transreal, relational operators: *less than* ($<$), *equal to* ($=$), *greater than*, ($>$). All combinations of these operators and logical *negation* (!) are distinct and are, therefore, useful. Distinctness is proved in the Appendix. Nullity is the only transreal number which compares not less than zero, not equal to zero and not greater than zero so transreal arithmetic does not require an *unordered* operator (?). But this counter example proves that *unordered* is logically redundant in IEEE 754 arithmetic.

Let us digress into a brief discussion of the meta theory of mathematics and computer science because this may be of interest to historians and philosophers of science, as well as to those mathematicians and scientists who use meta theory to direct their own research. What prevented computer scientists from noticing that the *unordered* operator is redundant? Those who did not read the standard would not be aware of the claim that the floating-point, relational operators are mutually distinct and would not be aware of the contrary evidence that only 22 relations are defined, where combinatorics *requires* 32. Those scientists would, however, be faced with the difficulty of using the operators but resolving their difficulties would require them to direct their attentions away from the object of their study to a study of the IEEE 754 standard. It is entirely understandable that many would regard this as an unacceptable distraction. But what of those who did read the standard? How can such an elementary error escape the many computer scientists who have worked on the standard and who have produced formal proofs of its correctness? An historian would ask them and read their notes, a philosopher might hypothesise that division by zero is an exceptional case that is handled *sui generis* so that it is not subjected to the usual tests of correctness. Quite simply, most scientists do not know how to divide by zero so they cannot test proposed properties of division by zero. To guard against this class of failures, one might explicitly propose that all *sui generis* cases should be examined with a view to embracing them in a total theory. We do this, here, by proposing the *design goal of totality*.

Our position is that the *unordered* operator of IEEE 754 floating-point arithmetic is logically redundant. If it has any role, it can only be in IEEE 754's model of exception handling but, as transreal arithmetic demonstrates, division by zero need not be taken as an exception so IEEE 754's error handling is redundant, in this case, making the *unordered* operator entirely redundant.

Let us count the number of compound IEEE 754 floating-point operations that are made redundant. IEEE 754 has 22 compound operators, transreal arithmetic and trans-floating-point arithmetic have 16 so at least $22 - 16 = 6$ of the operators are redundant but of the 16 transreal operators two, *Epsilon* and *Not Epsilon*, are not floating-point operators so trans-floating-point arithmetic has only 14 floating-point, relational operators. Hence $22 - 14 = 8$ of the IEEE floating-point, relational operators are redundant.

Observe that transreal arithmetic takes *less than, equal to* and *greater than* as total operators that apply to all transreal numbers where IEEE 754 introduces a special operator, *unordered*, to handle the *sui generis* category of NaNs. The IEEE 754 standard [1] fails, here, because it does not observe the *design goal of totality*. This failure gives rise to further exceptional cases which are tackled in a revision of the standard. The revised IEEE 754 standard [2] provides a special operator, $\mathrm{totalOrder}(x, y)$, that imposes a total order,

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

$x \preceq y$, on those x, y that are in canonical form. Taking NaN_i with the sign bit zero and $-\mathrm{NaN}_i$ with the sign bit non-zero and r_j a positive, represented, real number, we have a total ordering $-\mathrm{NaN_i} \prec -\infty \prec -r_j \prec 0 \prec r_j \prec \infty \prec \mathrm{NaN}_i$ for specified i, j. This produces a correct, total ordering of the bit patterns but, perversely, it makes all of the abstract objects encoded by the bit patterns unordered. Recall that IEEE 754 does not distinguish between NaNs with different signs so the bit patterns $-\mathrm{NaN_i}$ and $\mathrm{NaN_i}$ provide two representatives of each abstract $\mathrm{NaN_i}$. Now any real or infinite number n_j is both greater and less than some abstract $\mathrm{NaN_i}$ because $-\mathrm{NaN_i} \prec n_j \prec \mathrm{NaN_i}$ and any abstract $\mathrm{NaN_i}$ is both greater and less than itself, both of which cases are proved by $-\mathrm{NaN_i} \prec \mathrm{NaN_i}$. Taking these cases together, every abstract object represented by IEEE 754 bits is totally *un*ordered. This is perverse and having a function called *totalOrder* produce a total unorder is both ironic and anti-mnemonic for programmers. It succeeds only at the level of bit patterns, not at the level of abstract objects, so it is an example of data anti-abstraction and one which is mandated by the standard!

Transreal arithmetic does not have a total order operator: nullity is the only unordered number and all other transreal numbers are totally ordered by the transreal, relational operators. This entire order is encoded by these same relational operators so no additional information is needed. If an end user wants a particular total order, he of she is at liberty to put nullity anywhere in the sequence; we generally recommend taking it first so that the unique number nullity is processed before arbitrarily many of the ordered numbers.

Now let us consider the cognitive burden on programmers, further to the exceptions, anti-mnemonic and data anti-abstraction noted above.

The ordering of transreal numbers is shared with trans-floating-point arithmetic and trans-two's-complement arithmetic [3] so the same relational operators and control exceptions, such as *inexact* result, apply to both systems. The programmer does not have to learn separate relations and control exceptions for each system. Even better, the ordering relations are just the ordering relations of real arithmetic so the programmer, who is familiar with real arithmetic, need only learn that nullity is the uniquely unordered number.

The transreal, relational operators are orthogonal in the sense that every combination of operators is allowed. Hence the programmer does not have to learn exceptions. Can the reader say which 10 relational operators are not supported by IEEE 754 floating-point arithmetic or explain the circumstances in which $-f \neq 0 - f$? How much work will it take the reader to answer these questions and what profit is there in that labour? What is the cost to society in demanding such unproductive labour of programmers?

Transreal arithmetic is total and can be used to totalise certain functions so that they have no exceptions. For example we may totalise the hardware square-root function so that the square root of a negative number returns nullity. In this case the number nullity is being used as a flag but as nullity is absorptive over the transreal operations of arithmetic, so that all sums, differences, products and quotients of nullity are nullity, the flag, nullity, will propagate. Its meaning, in this model, is that there is no extended-real number that is the square root of a negative number. As nullity is not an extended-real number, it carries this information faithfully. We expect that similar arguments can be made for all of the real functions of elementary algebra.

Of course transreal arithmetic cannot be used to totalise all functions. For example the function $f(a, b) = c$ that returns some one transreal number, c, such that $a <= c$ and $c <= b$ produces no result for $f(1, 0)$. Such mathematical functions can be totalised by operating on sets so that, in this example, the solution set is empty. Another approach is to use a separate, say Boolean, flag to indicate whether the result of a function is valid or not. This is equivalent to using a hardware *invalid opearation* exception but it remains to be established that the transreal versions of any of the IEEE 754 floating-point functions do have such exceptions.

The reader is faced with a paradigm shift. The reader was educated at a time when division by zero was generally considered impossible. Consequently the reader was taught a partial arithmetic that fails on division by zero and partial mathematics that fail similarly. Working in that paradigm the reader has little guidance on how to develop a total arithmetic and so is thrown back on a series of *ad hoc* decisions; each time an infelicitous decision is made, further *ad hoc* additions must be made to try to correct them. By contrast transreal arithmetic is now available. It supports division by zero, is total, and is being developed, systematically, into a transmathematics. If the reader makes the paradigm shift to the new system, he or she will work from the basis of a total system and will have the systematic guidance of mathematical derivations to develop total computing systems. This paper offers a deal: accept division by zero and gain a simpler programming system with up to twice the accuracy of IEEE 754 floating-point arithmetic or reject the deal and carry on as now.

V. Conclusion

IEEE 754 floating-point arithmetic is widely used but it is based on an *ad hoc* totalisation of real arithmetic with many infelicities, some of which are discussed above. From a mathematical point of view, the worst infelicity is that the equality operator is not reflexive so that $x_1 = x_2$ is true for some unequal bit patterns x_1, x_2 and is false for some equal bit patterns x_1, x_2. It is certainly possible to maintain a consistent semantics in the face of this and related difficulties but it is not easy to do. The practical difficulty of achieving consistency is demonstrated by inconsistent floating-point behaviour between commercially important programming languages that adhere to the relevant programming language standards. A longer version of this paper could demonstrate this fact with source code and could suggest software ameliorations based on the semantics of transreal arithmetic, thereby demonstrating its utility as a superior total arithmetic.

The IEEE 754 floating-point arithmetic standard acknowledges that some failures of interoperability are caused by the Not-a-Number, NaN, elements. We have proved that the standard is wrong when it says that its basic relational operators - *less than, equal to, greater than, unordered* - are mutually exclusive. Specifically we prove that the *unordered* relation is logically redundant, having utility only in the IEEE 754 model of error handling; then we show that this error model is redundant when transreal arithmetic, which has no NaNs, is used as the basis of floating-point arithmetic. Thus transreal arithmetic simplifies the relational

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

operators, simplifies programming and removes an entire class of errors; all of which supports our view that trans-floating-point arithmetic is a superior model of floating-point arithmetic. We expect that trans-floating-point arithmetic will be better suited to safety critical applications, especially where formal verification of code is mandatory.

Transreal arithmetic is controversial but it offers both practical and theoretical advantages. For its proponents, the transreal numbers $-\infty = -1/0$, $\infty = 1/0$ and $\Phi = 0/0$ are all valid numbers with well defined mathematical properties and well defined semantics in mathematical models of practical systems. We emphasise just two of its practical advantages. Firstly transreal arithmetic uses only the ordinary relational operators for *less than*, *equal to* and *greater than*, taking Φ as the uniquely unordered number. When implemented as a computer arithmetic, this reduces the number of relational operators, as compared to IEEE 754 arithmetic, and removes all exceptions from them. This makes programming both simpler and safer, with fewer cases to verify. Secondly using transreal arithmetic as a basis for floating-point arithmetic would provide numerical computations with up to twice the accuracy of IEEE 754 floating-point arithmetic. These practical advantages ought to be of scientific and commercial interest.

Finally we propose that there is a single, conceptual failure in the design of IEEE 754 floating-point arithmetic that explains all of its infelicities: the standard fails to impose totality and instead attempts to impose solutions to each of the consequences of partiality. As there are infinitely many such consequences, all revisions of the standard will fail, until totallity is accepted as a design goal. We observe that this failure is almost universal in software design so adopting the *design goal of totality* would improve the performance and reliability of almost all software.

Appendix A
Transreal Relational Operators

There are three basic, transreal, relational operators: *less than* ($<$), *equal to* ($=$), *grater than*, ($>$). These operators are mutually exclusive so they can be combined in $2^3 = 8$ ways. All 8 combinations are distinct and meaningful, including the empty operator with no occurrences of the basic operators. All 8 combinations can be combined with the logical negation operator ($!$). This yields $2 \times 2^3 = 2^4 = 16$ distinct and meaningful operators. The multiplication table for each operator is given here.

The relational operators can be formalised as production rules of the form $a \bullet b \rightarrow c$, where $\bullet$ is the operator. Hence "$a \bullet b$" is replaced by "c". The empty operator is indicated by epsilon (ϵ) so "$a\epsilon b$" is identical to "ab" whence the empty operator implements the identity concatenation $ab \rightarrow ab$. This is shown in the first multiplication table, entitled *Epsilon.* This operator occurs, trivially, in all written languages, including computer languages. Combining the empty operator with the logical negation operator yields "$a!\epsilon b$" which is identical to "$a!b$" and, following custom, we take the operator "!" as a unary, right associative operator, so that, for example, "X!T" is replaced by "XF" where T stands for True, F stands for False and X stands for an arbitrary symbol. This is shown in the second multiplication table, entitled *Not Epsilon.* This operator, with a possibly different glyph, occurs in most high-level, computer languages. The remaining multiplication tables are truth tables. The labels on the rows and columns indicate the arguments: negative infinity ($-\infty$), an arbitrary real number (r_i), positive infinity (∞), nullity (Φ). As usual T stands for True and F stands for False. In a departure from the usual notation, an asterisk (*) stands for a conditional truth value. For example, in the third table, entitled *Less*, the asterisk in the row labeled r_1 and column labeled r_2 is to be replaced by the truth value of $r_1 < r_2$, and similarly in the other tables. This recruitment of the real relation, less than, to define the corresponding transreal relation, is a context-sensitive reading of the symbol $<$. Computer scientists are generally comfortable with context-sensitive readings but many mathematicians regard them as an abuse of notation; even so, such notations are very common and are easily understood.

It can be seen, by inspection, that the multiplication tables are distinct. The labour of inspecting the tables can be reduced by exploiting symmetries. It is sufficient to notice that the first two elements, respectively FT, TF, FF of the first row of the tables *Less, Equal, Greater* are distinct and, similarly, TT, FT, TF of *Less or Equal, Less or Greater, Greater or Equal* are distinct.

Epsilon

ϵ	b
a	ab

Not Epsilon

$!\epsilon$	F	T
a	aT	aF

Less

$<$	$-\infty$	r_1	∞	Φ
$-\infty$	F	T	T	F
r_1	F	*	T	F
∞	F	F	F	F
Φ	F	F	F	F

Equal

$=$	$-\infty$	r_1	∞	Φ
$-\infty$	T	F	F	F
r_1	F	*	F	F
∞	F	F	T	F
Φ	F	F	F	T

Greater

$>$	$-\infty$	r_1	∞	Φ
$-\infty$	F	F	F	F
r_1	T	*	F	F
∞	T	T	F	F
Φ	F	F	F	F

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Less or Equal

$<=$	$-\infty$	r_1	∞	Φ
$-\infty$	T	T	T	F
r_1	F	*	T	F
∞	F	F	T	F
Φ	F	F	F	T

Less or Greater

$<>$	$-\infty$	r_1	∞	Φ
$-\infty$	F	T	T	F
r_1	T	*	T	F
∞	T	T	F	F
Φ	F	F	F	F

Greater or Equal

$>=$	$-\infty$	r_1	∞	Φ
$-\infty$	T	F	F	F
r_1	T	*	F	F
∞	T	T	T	F
Φ	F	F	F	T

Less or Equal or Greater

$<=>$	$-\infty$	r_1	∞	Φ
$-\infty$	T	T	T	F
r_1	T	T	T	F
∞	T	T	T	F
Φ	F	F	F	T

Not Less

$!<$	$-\infty$	r_1	∞	Φ
$-\infty$	T	F	F	T
r_1	T	*	F	T
∞	T	T	T	T
Φ	T	T	T	T

Not Equal

$!=$	$-\infty$	r_1	∞	Φ
$-\infty$	F	T	T	T
r_1	T	*	T	T
∞	T	T	F	T
Φ	F	F	F	T

Not Greater

$!>$	$-\infty$	r_1	∞	Φ
$-\infty$	T	T	T	T
r_1	F	*	T	T
∞	F	F	T	T
Φ	T	T	T	T

Not Less or Equal

$!<=$	$-\infty$	r_1	∞	Φ
$-\infty$	F	F	F	T
r_1	T	*	F	T
∞	T	T	F	T
Φ	T	T	T	T

Not Less or Greater

$!<>$	$-\infty$	r_1	∞	Φ
$-\infty$	T	F	F	T
r_1	F	*	F	T
∞	F	F	T	T
Φ	T	T	T	T

Not Greater or Equal

$!>=$	$-\infty$	r_1	∞	Φ
$-\infty$	F	T	T	T
r_1	F	*	T	T
∞	F	F	F	T
Φ	T	T	T	F

Not Less or Equal or Greater

$!<=>$	$-\infty$	r_1	∞	Φ
$-\infty$	F	F	F	T
r_1	F	F	F	T
∞	F	F	F	T
Φ	T	T	T	F

ACKNOWLEDGMENT

The author would like to thank the members of Transmathematica for many helpful discussions.

REFERENCES

[1] Ieee standard for binary floating-point arithmetic. 1985.
[2] Ieee standard for floating-point arithmetic. 2008.
[3] James A.D.W. Anderson. Perspex machine xi: Topology of the transreal numbers. In S.I. Ao, Oscar Castillo, Craig Douglas, David Dagan Feng, and Jeong-A Lee, editors, *IMECS 2008*, pages 330–33, March 2008.
[4] James A.D.W. Anderson. Evolutionary and revolutionary effects of transcomputation. In *2nd IMA Conference on Mathematics in Defence*. Institute of Mathematics and its Applications, Oct. 2011.
[5] James A.D.W. Anderson, Norbert Völker, and Andrew A. Adams. Perspex machine viii: Axioms of transreal arithmetic. In Longin Jan Lateki, David M. Mount, and Angela Y. Wu, editors, *Vision Geometry XV*, volume 6499 of *Proceedings of SPIE*, pages 2.1–2.12, 2007.

ISBN: 978-988-19252-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)