The Scheme of Things:
Exact and Inexact Numbers

William Clinger
Semantic Microsystems, Inc.

Numbers are one of the greatest achievements of humanity. They are the product of many thousands of years of effort by the best minds of history—and prehistory. That effort continues, for our concept of numbers is, and will remain, unfinished.

Numbers have always been central to computing, and hence to programming languages. The "numbers" supported by a programming language, however, often behave quite differently from the mathematical concepts on which they are modelled. Many of these differences are motivated by concern for efficient execution on affordable hardware, which is important for a programming language but meaningless for mathematics. The most obvious example is the "integer" arithmetic provided by most languages, which in practice usually turns out to be arithmetic modulo some power of two.

Another example is the rigid and artificial distinction between "integer" and "real" numbers in many languages. Historically, this distinction seems to have arisen in programming languages through a process that we might scorn today as a violation of the principle of representation independence. At the time FORTRAN was introduced, however, it seemed natural to think that if a language's implementation will use two different representations for numbers, then the language must use two different kinds of numbers. The reason that two different representations were required by the implementation is still valid today: One representation, *floating point*, was and is the best representation known for convenient and efficient calculation with approximations to the real numbers over a wide range of values. The other representation, which varies depending on the hardware, was and is best for efficient and exact calculations with integers over the small range of values that can be represented within a machine word or register. It is hard to fault FORTRAN's decision that programmers, when declaring a variable that will be used to hold numeric values, must also declare the representation that will be most efficient for the intended use of that variable. For that matter, FORTRAN deserves respect rather than contempt for seeking to make this requirement less burdensome by allowing programmers to omit the declaration when they follow established mathematical practice in choosing the names of their variables.

Of course, a good idea in FORTRAN's time isn't necessarily a good idea today. How should numbers be organized in a modern programming language? Should they be organized as a collection of data types, with well-defined coercions from the more specialized types such as integers into more general types such as reals, or should numbers be a single data type (leaving open the possibility of several distinct internal representations)?

This question is analogous to a question that arises in the foundations of mathematics. Consider, for example, how the field of complex numbers can be constructed from set theory. The first step is to define the natural numbers. Following John von Neumann, we can take the empty set as zero and define the successor function *succ* by

$$succ(n) = n \cup \{n\}.$$

Hence one will be the set containing zero, two will be the set containing both zero and one, and so

on: each natural number is represented as the set consisting of all the previous natural numbers:

$$
\begin{aligned}
0 &= \{\,\} \\
1 &= \{0\} = \{\{\,\}\} \\
2 &= \{0,1\} = \{\{\,\},\{\{\,\}\}\} \\
3 &= \{0,1,2\} = \{\{\,\},\{\{\,\}\},\{\{\,\},\{\{\,\}\}\}\} \\
4 &= \{0,1,2,3\} = \{\{\,\},\{\{\,\}\},\{\{\,\},\{\{\,\}\}\},\{\{\,\},\{\{\,\}\},\{\{\,\},\{\{\,\}\}\}\}\} \\
5 &= \ldots
\end{aligned}
$$

We can then define the set of natural numbers recursively as the closure of $\{0\}$ under the successor operation, and write it as $\omega = \{0,1,2,3,4,5,\ldots\}$. This representation, while arbitrary, has some nice properties. Among the most important for mathematics is that it generalizes nicely to infinite numbers—can you guess what $\omega + 1$ is? Equality of natural numbers coincides with equality of numbers considered as sets, and the less-than relation $<$ coincides with the subset relation. We can go on to define addition and multiplication in the usual recursive way.

The next step is to define the integers. By taking 0 as the plus sign and 1 as the minus sign, we can represent the integers as the set

$$Z = (\{0\} \times \omega) \cup (\{1\} \times (\omega - \{0\}))$$

and then define operations on integers in the usual way. Next come the rationals, which we can represent as $Q = (Z \times (Z - \{\langle 0,0\rangle\}))/\equiv$, where $\equiv$ is the equivalence relation defined by

$$\langle a,b\rangle \equiv \langle c,d\rangle \text{ if and only if } ad = bc.$$

After defining the usual operations on the rationals, including the less-than relation $<$, we can represent the reals as the set of Dedekind cuts. A Dedekind cut is a nonempty proper subset of the rationals that is closed under $<$ and has no greatest element:

$$
\begin{aligned}
R = \{A \subseteq Q \mid {} & \exists q\ q \in A \ \wedge\ \exists q \in Q\ q \notin A \\
& \wedge\ \forall a \in A\ (\forall q \in Q\ q < a \Rightarrow q \in A \ \wedge\ \exists a' \in A\ a < a')\}
\end{aligned}
$$

It is easy to define the usual operations on the reals. Finally we can represent the complex numbers in rectangular coordinates as $C = R \times R$ and go on to define operations on them.

We have represented the complex numbers C using the real numbers R, which are represented using the rational numbers Q, which are represented using the integers Z, which are represented using the natural numbers ω. These representations are satisfactory in most respects, but there's a problem. The problem is that we learned in school that the natural numbers are a subset of the integers, that the integers are a subset of the rationals, that the rationals are a subset of the reals, and that the reals are a subset of the complex numbers. None of that is true of the numbers we have built. There are two ways to fix our construction. One is to define a set of coercion functions

$$
\begin{aligned}
\alpha_{\omega,Z} &: \omega \to Z \\
\alpha_{Z,Q} &: Z \to Q \\
\alpha_{Q,R} &: Q \to R \\
\alpha_{R,C} &: R \to C \\
\alpha_{\omega,Q} &= \alpha_{\omega,Z} \circ \alpha_{Z,Q} \\
\alpha_{\omega,R} &= \alpha_{\omega,Q} \circ \alpha_{Q,R} \\
\alpha_{\omega,C} &= \alpha_{\omega,R} \circ \alpha_{R,C} \\
\alpha_{Z,R} &= \alpha_{Z,Q} \circ \alpha_{Q,R} \\
\alpha_{Z,C} &= \alpha_{Z,R} \circ \alpha_{R,C} \\
\alpha_{Q,C} &= \alpha_{Q,R} \circ \alpha_{R,C}
\end{aligned}
$$

that map each natural number to the corresponding integer, and so on, and then to adopt the following ideology:

> **Ideology 1.** Henceforth, whenever we say that A is a subset of B, where A and B are both among ω, Z, Q, R, and C, what we really mean is that the image of A under the coercion function $\alpha_{A,B}$ is a subset of B. Whenever we say that $b \in B$ is a member of A, what we really mean is that b is the image of some $a \in A$ under $\alpha_{A,B}$.

A different solution is to regard the complex numbers C (or perhaps some other number system such as the reals) as the all-encompassing set of true numbers, and to regard the sets ω, Z, Q, and R as mere scaffolding used in the construction of C. This requires us to adopt the following ideology:

> **Ideology 2.** Henceforth, whenever we speak of A, where A is ω, Z, Q, or R, what we really mean is the image of A under $\alpha_{A,C}$.

The second ideology seems to be the one most often adopted in mathematics, and its analogue for programming languages is the ideology adopted by Scheme: There is an all-encompassing set of numbers, and all the various number systems are true subsets of it. Most programming languages adopt an analogue of the first ideology: The various number systems define disjoint sets of numbers, but these sets are related by coercion functions.

Common Lisp, for example, requires at least three disjoint numeric types: `rational`, `float`, and `complex`. (Oddly, the type `integer` is a subtype of `rational`.) No `rational` is ever a `float` or a `complex`, and no `float` is ever a `complex`. The coercion procedures are `rational`, `float`, and `complex`.

Scheme, by contrast, requires that every rational be both a real and a complex, and that every real be a complex. Likewise every integer must also be a rational, a real, and a complex number. Coercion procedures are unnecessary, since they would be identity functions.

In its use of a single numeric type, Scheme follows the lead of programming languages such as APL. Scheme's contribution is its division of that type into the subtypes of *exact* and *inexact* numbers. These subtypes are disjoint, but unlike the disjoint numeric subtypes of other programming languages they do not correspond to any of the number systems constructed above. They are motivated by pragmatics rather than by mathematics.

The pragmatic motivation is based on an interesting property of the representations for numbers that have been found to be most useful in programming languages. These representations are known as *fixnums*, which can represent small integers; *bignums*, which can represent integers of any size; *ratnums*, which correspond to pairs of bignums or fixnums and can represent rationals of any size; *flonums*, which are floating point approximations to real numbers within a large but fixed range; and *complexnums*, which correspond to pairs of flonums and can approximate complex numbers whose rectangular components are within the range that can be represented by flonums. These and other representations fall neatly into two classes according to their behavior with respect to the common arithmetic operations such as addition, subtraction, multiplication, and division. One class consists of representations such as flonums that are inherently approximate in the sense that the flonum result of adding two flonums is unlikely to represent the exact result of adding the numbers represented by those flonums. The other class consists of representations such as fixnums and bignums that are exact in the sense that the fixnum or bignum obtained by adding two fixnums does indeed represent the exact result of adding the numbers represented by those fixnums. Scheme's distinction between exact and inexact numbers is an abstract recognition of this important distinction between these two classes of representations. While it complicates the language, the distinction yields four pragmatic benefits: efficiency, predictability, reliability, and flexibility of implementation.

Inexactness is a contagious property. Adding an exact number and an inexact number yields an inexact number. This improves efficiency because arithmetic operations on the representations used for inexact numbers (flonums and complexnums) are faster than on ratnums or pairs of ratnums, which are the representations for exact numbers that would probably have to be used if the result were not represented as a flonum or complexnum.

Of course, this efficiency could be obtained without making the distinction between exactness and inexactness explicit in the programming language. By making the distinction explicit and stating the conditions under which inexact results can be obtained, Scheme helps programmers to understand and anticipate the behavior of their programs. The distinction between exact and inexact numbers thus helps to make programs more predictable. This predictability is greatly weakened, however, by the flexibility that Scheme gives to implementations as discussed below.

The concept of exactness also helps to catch bugs. For example, it seems unlikely that a programmer would deliberately use an inexact integer as a vector index. By defining this to be an error, Scheme allows implementations to detect and report it. Requiring exact integers as indexes also makes vector and string manipulation more efficient in Scheme than in most other languages that use a single numeric type, because procedures such as `vector-ref` and `string-ref` do not have to be prepared to convert flonum representations of inexact integers into the representation required by the addressing hardware.

Finally, the abstractness of the distinction between exact and inexact numbers offers considerable flexibility to implementors of Scheme. A typical implementation of Scheme might represent exact numbers as fixnums, bignums, and ratnums, and represent inexact numbers as flonums or complexnums. An implementation could, however, represent all exact numbers as ratnums and all inexact numbers as complexnums. At the other extreme, an implementation could represent all exact numbers as fixnums and all inexact numbers as flonums, since Scheme currently does not require that complex numbers be supported, does not require that non-integers be supported as exact numbers, and does not even require that integers be supported as exact numbers outside any limits imposed by the implementation on the sizes of vectors, strings, and proper lists. While this flexibility is bad for predictability and portability, it is valuable for research, experimentation, and applications such as embedded systems.

* * *

[1] Willard Van Orman Quine. *Set Theory and Its Logic, Revised Edition.* Harvard University Press, 1969.

[2] Jonathan Rees and William Clinger [editors]. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* **21**, 12, December 1986, 37–79.