IDENTITY CRISIS

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 86, Number 6
November–December, 1998
pages 508–512

IDENTITY CRISIS

Brian Hayes

Suppose I had once borrowed your boat and, secretly, replaced each board with a similar but different one. Then, later, when I brought it back, did I return your boat to you? What kind of question is that? It's really not about boats at all, but about what people mean by "same."

—Marvin Minsky, *The Society of Mind*

I have heard of a touchy owner of a yacht to whom a guest, on first seeing it, remarked, 'I thought your yacht was larger than it is'; and the owner replied, 'No, my yacht is not larger than it is'.

—Bertrand Russell, "On Denoting"

The equal sign seems to be a perfectly innocent bit of mathematical notation. We all know exactly what it means. The symbol "=" is the fulcrum of a balance. It declares that the things on either side of it, whatever they may be, are equivalent, identical, alike, indistinguishable, the same. What could be clearer? Although an equation may be full of mystery—I can't explain what $e^{i\pi} = -1$ really means—the enigma lies in the two objects being weighed in the balance; the equal sign between them appears to be totally straightforward.

But equality isn't as easy as it looks. Sometimes it's not at all obvious whether two things are equal—or even whether they are two things. In everyday life the subtle ambiguities of identity and equality are seldom noticed because we make unconscious allowances and adjustments for them. In mathematics they cause a little more trouble, but the place where equality gets really queer is in the discrete, deterministic and literal-minded little world of the digital computer. There, the simple act of saying that two things are "the same" can lead into surprisingly treacherous territory.

What follows is a miscellaneous collection of problems and observations connected in one way or another with the concepts of equality and identity. Some of them are mere quibbles over the meaning of words and symbols, but a few reflect deeper questions. The difficulty of defining equality inside the computer may even shed a bit of light on the nature of identity in the physical world we think we live in.

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org.*

Some Are More Equal Than Others

There's an old story about the mathematician who sets out to learn a computer programming language such as FORTRAN or C. Everything goes swimmingly until she comes to the statement $x = x + 1$, whereupon she concludes that computer programming is mathematical nonsense.

Of course this story is just a programmer's joke at the expense of mathematicians. I would respond in the same spirit by suggesting that a mathematician would be better equipped than anyone else to solve the "equation" $x = x + 1$. Obviously x is equal to $\aleph_0$, the infinite ordinal number, which has just the property that $\aleph_0 = \aleph_0 + 1$.

In truth, $x = x + 1$ is not an equation at all in FORTRAN or C, because the symbol "=" is not an equal sign in those languages. It is not a relational operator, comparing two quantities, but an assignment operator, which *manufactures* equality. When an assignment statement is executed, whatever is on the left of the "=" sign is altered to make it equal to the value of the expression on the right. The semantics of this operation are altogether different from those of testing two things for equality. As it happens, the semantics of assignment introduce certain troublesome characteristics into computer programs, which I shall have occasion to mention again below.

To avoid confusion between equality and assignment, many programming languages choose different symbols for the two operations. For example, Algol and its descendants write ":=" for assignment. In all that follows the symbol "=" will mean only equality (whatever that means).

By the way, the "=" notation was invented by Robert Recorde (1510–1558). He chose two parallel lines as a symbol of mathematical equality "because noe 2 thynges can be moare equalle."

Equality of Beans

Just what *does* it mean for two numbers to be equal? Note that this is a question about numbers, not numerals, which are representations of numbers. The numeral 5 in base 10 and the numeral 11 in base 4 and the Roman numeral V all

represent the same number; they are all equal. Numbers can even be represented by piles of beans, which form a unary, or base-1, notation.

For small enough hills of beans, most people can judge equality at a glance, but computers are no good at glancing. If you want to teach a computer to test bean piles for equality, you'll need an algorithm. There's a very simple one that works on piles of any finite size. The only mathematical skill the computer needs is the ability to count from 0 up to 1: It has to be able to recognize an empty pile and to choose a single bean from a nonempty pile. Then it can determine the equality of two piles by following these rules: First, check the piles to see if they are empty. If both piles are empty, they are obviously equal. If one pile is empty and the other isn't, the piles are unequal. If neither pile is empty, remove one bean from each pile and repeat the whole procedure. Since the piles are finite, at least one of them must eventually be emptied, and so the algorithm will always terminate.

This method is loosely based on a scheme devised a century ago by the Italian mathematician Giuseppe Peano, who formulated a set of axioms for arithmetic in the natural numbers (also known as the counting numbers, or the nonnegative integers). Peano's method can be generalized, though awkwardly, to the set of all integers, including negative whole numbers. But the algorithm doesn't work for the real numbers (those that make up the continuum of the real number line). Indeed, the very concept of equality among the reals is perplexing. For the rational numbers, which form a subset of the reals, equality is no problem. Since every rational can be represented as a fraction reduced to lowest terms, two rationals are equal if their numerators are equal and their denominators are equal. The trouble is, almost all the reals are *irrational*; if you choose a point at random along the real number line, the probability of landing on a rational is zero. And no finite process can show that two arbitrary irrational numbers are equal.

Irrational numbers are usually represented as nonrepeating decimals, such as the familiar 3.14159 for the first few digits of π. Because the pattern of digits never repeats, matching up two irrational numbers digit by digit will never prove them equal; there are always more digits yet to be checked.

Writing numbers in decimal form has another pitfall as well: A single real number can have multiple decimal expansions, which are mathematically equivalent but don't look at all alike. The problem comes up even with rational numbers. A case in point is the pair of values 0.999... and 1.000... (where the three dots signify an infinitely repeated pattern of digits). These numerals have not one digit in common, and yet they denote exactly the same value; there is not the least smidgen of difference between them. (If you doubt this, consider that 0.333... + 0.333... + 0.333... = 0.999..., but ⅓ + ⅓ + ⅓ = 1.) Thus some real numbers look alike but can't be proved equal, while others are equal but look very different.

Same Difference

Real numbers are creatures of mathematics, not computer science. Although some programming languages offer a data type named "real," the numbers of this type are quite *un*real. They are "floating-point" numbers, which only approximate a continuous distribution. The floating-point format is much like scientific notation, in which a number is represented by a mantissa and an exponent. Only a finite number of bits are reserved for the mantissa and exponent, and so the numbers are limited in both precision and range.

One big advantage of floating-point arithmetic is that you never have to wait forever. When floating-point values stand in for reals, questions about equality are always answered promptly. On the other hand, the answers may well be wrong.

In your favorite programming language, calculate the square root of 2 and then square the result. I've just tried this experiment with an old programmable calculator, which reports the answer as 1.999999999. Interpreted as an approximation to the real number 1.999..., this result is not an error. It's just as correct as 2.000000000, which is also an approximation. The problem is that the machine itself generally cannot recognize the equivalence of the two alternative answers. Suppose a program includes the conditional statement:

if $(\sqrt{2})^2 = 2$
then let there be light
else annihilate the universe

If this program happens to be running on my HP-41C, we're all in trouble.

There are alternatives to floating-point arithmetic that avoid these hazards. Symbolic-mathematics systems such as Maple and Mathematica get the right answer by eschewing numerical approximations; in effect, they define the square root of 2 as "the quantity that when squared is equal to 2." A few programming languages provide exact rational arithmetic. And there have been various schemes

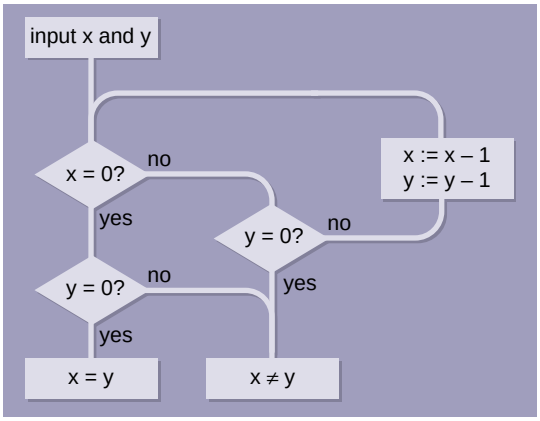


Figure 1. Peano's algorithm establishes the equality of two natural numbers, or nonnegative integers.

for calculating with approximations to real numbers that grow more digits on demand, potentially without limit. Nevertheless, most numerical computation is done with conventional floating-point numbers, and a whole subdiscipline of numerical analysis has grown up to teach people how to cope with the errors.

Programmers are sometimes advised not to compare floating-point values for exact equality, but rather to introduce a small quantity of fudge. Instead of computing the relation $x = y$, they are told to base a decision on the expression $|x - y| < \varepsilon$, where the straight brackets denote the absolute-value operation, and ε is a small number that will cover up the imprecision of floating-point arithmetic. This notion of approximate equality is good enough for many purposes, but it has a high cost. Equality loses one of its most fundamental properties: It is no longer a transitive relation. In the presence of fudge, you can't count on the basic principle that if $x = y$ and $y = z$, then $x = z$.

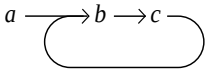
Some Are Less Equal Than Others

Numbers aren't the only things that can be equal or unequal. Most programming languages also have equality operators for other simple data objects, such as alphabetic characters; thus a = a but a ≠ b. (Whether a = A is a matter up for debate.)

Sequences of characters (usually called strings) are also easy to compare. Two strings are equal if they consist of the same characters in the same sequence, which implies the strings also have the same length. Hence an equality operator for strings simply marches through the two strings in parallel, matching up the characters one by one. Certain other data structures, such as arrays, are handled in much the same way.

But one important kind of data structure can be problematic. The most flexible way of organizing data elements is with links, or pointers, from one item to another. For example, the symbols *a*, *b* and *c* might be linked into the list $a \rightarrow b \rightarrow c \rightarrow nil$, where *nil* is a special value that marks the end of a chain of pointers. Comparing two such structures for equality is straightforward: Just trace the two chains of pointers, and if both reach *nil* at the same time without having encountered any discrepancies along the way, they are identical.

The pointer-following algorithm works well enough in most cases, but consider this structure:



An algorithm that attempts to trace the chain of pointers until reaching *nil* will never terminate, and so structural equality will never be decided. This problem can be solved—the workaround is to lay down a trail of breadcrumbs as you go, and stop following the pointers as soon as you recognize a site you've already visited—but the technique is messy.

There's something else inside the computer that's remarkably hard to test for equality: programs. Even in the simplest cases, where the program is the computational equivalent of a mathematical function, proving equality is a challenge. A function is a program that accepts inputs (called the arguments of the function) and computes a value, but does nothing else to alter the state of the computer. The value returned by the function depends only on the arguments, so that if you apply the function to the same arguments repeatedly, it always returns the same value. For example, $f(x) = x^2$ is a function of the single argument x, and its returned value is the square of x.

A given function could be written as a computer program in many different ways. At the most trivial level, $f(x) = x^2$ might be replaced by $f(y) = y^2$, where the only change is to the name of the variable. Another alternative might be $f(x) = x \times x$, or perhaps $f(x) = \exp(2 \log(x))$. It seems reasonable to say that two such functions are identical if they return the same value when applied to the same argument. But if that criterion were to serve as a test of function equality, you would have to test all possible arguments within the domain of the function. Even when the domain is not infinite, it is often inconveniently large. The alternative to such an "extensional" test of equality is an "intensional" test, which tries to prove that the texts of the two programs have the same meaning. Fabricating such proofs is not impossible—optimizing compilers do it all the time when they substitute a faster sequence of machine instructions for a slower one—but it is hardly a straightforward task.

When you go beyond programs that model mathematical functions to those that can modify the state of the machine, proving the equality of programs is not just hard but undecidable. That is, there is no algorithm that will always yield the right answer when asked whether two arbitrary programs are equivalent. (For a thorough discussion of program equivalence, see Richard Bird's book *Programs and Machines*.)

One and the Same

We seldom notice it, but words such as "equal," "identical" and "the same" conceal a deep ambiguity. Consider this pair of sentences:

> *On Friday Alex and Baxter wore the same necktie.*
> *On Friday Alex and Baxter had the same teacher.*

These two instances of "the same" are not at all the same. Unless Alex and Baxter were yoked together at the neck on Friday, they wore two ties; but they had only one teacher. In the first case two things are alike enough to be considered indistinguishable, and in the second case there is just one thing, which is necessarily the same as itself. The two concepts are so thoroughly entangled that it's hard to find words to speak about them. Where confusion is likely I shall emphasize the distinction with the terms "separate but equal" and "selfsame."

When there's some uncertainty about whether two things are alike or are really just one thing, the usual strategy is to examine them (it?) more closely.

If you study the two neckties long enough, you're sure to find some difference between them. Even identical twins are never truly identical; when you get to know them better, you learn that one has a tattoo, and the other can't swim. (If all else fails, you can ask them; they know who they are.)

The strategy of looking harder until you spot a difference doesn't work as well inside the computer, where everything is a pattern of bits, and separate patterns really can be equal. Bits have no blemishes or dents or distinguishing features.

Another way to decide between the two kinds of sameness is through the rule of physics (or metaphysics) that a single object cannot be in two places at the same time, and two objects cannot occupy the same space at the same time. Thus all you have to do is bring Alex and Baxter together in the same room and check their neckwear.

The computational equivalent of this idea states that two objects are in fact the selfsame object only if they have the same address in memory. Thus two copies of an object can be distinguished, even though they correspond bit-for-bit, because they have different addresses. This is a practical method in widespread use, and yet it has certain unsatisfactory aspects. In the first place it assumes that the computer's memory is organized into an array of unique addresses, which is certainly the usual practice but is not the only possibility. Second, letting identity hinge on location means that an object cannot move without changing into something else. This idea that where you live is who you are contradicts everyday experience. It is also a fiction in modern computer systems, where data are constantly shuffled about by mechanisms such as virtual memory, caches and the storage-management technique called garbage collection; to maintain the continuity of identity, all of these schemes have to fool programs into thinking that objects don't move—a source of subtle bugs.

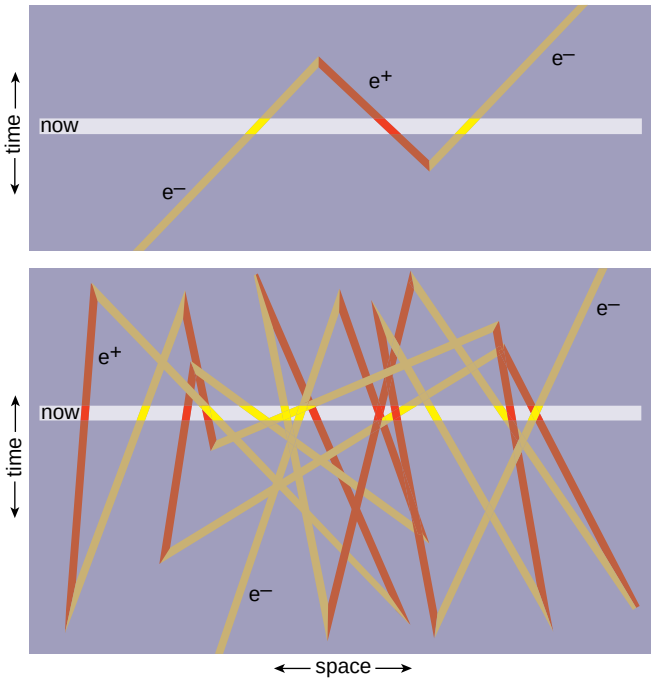


Figure 2. Why do all electrons look alike? Perhaps they are all the same electron! The event in the upper diagram is usually interpreted as an interaction of three particles: An electron and positron appear spontaneously, then the positron collides with a second electron and both particles are annihilated. But the event could be interpreted as the trace of a single particle that moves forward in time as an electron and backward as a positron. In the lower diagram a single particle with a tangled world line appears as multiple electrons and positrons.

More of the Same

There is a third way of exploring issues of identity, but it lies outside the realm of nondestructive testing. If Alex spills ketchup on his tie, does Baxter's tie also acquire a stain? If Baxter steps on his teacher's toe, does Alex's teacher have a sore foot? The principle being suggested here is that two things are the selfsame thing if changing one of them changes the other in the same way.

In computing, this process is encountered most often in the unexpected and unpleasant discovery that two variables are "aliases" referring to the same value or object. For example, if the variable designating Alex's grade-point average and the variable for Baxter's average both point to the same location in memory, then any change in one value will also be reflected in the other. This is probably not the desired behavior of the school's grading system.

In principle, deliberate alteration of memory contents could serve as a test of identity: Just twiddle the bits of an object and see if the corresponding bits of another object flip. But the test is not foolproof, particularly in a computer with multiple threads of execution. There's always the chance that the same change might be made coincidentally in two different places; two independent ketchup stains are not an impossibility. If coincidence seems too unlikely, consider that there might be a process running whose purpose is to synchronize two variables, checking one of them at frequent intervals and changing the other one to match. Or, conversely, a background process might undo any change to a variable, restoring the original value whenever it is modified. Under these conditions, a bit-flipping identity test might find that an object is not even equal to itself.

The distinction between separate-but-equal objects and a selfsame object can be crucially important. When I make a bank deposit, I'd strongly prefer that the amount be credited to my own selfsame account, rather than to the account of someone who is separate-but-equal to me—perhaps someone with the same name and date of birth. The standard practice for maintaining identity in these circumstances is to issue a unique identifying number. These are the numbers that rule so much of modern life, the ones you find on your bank statement, your driver's license, your credit cards. The same technique can be used internally by a computer program to keep track of data objects. For example, the programming language

Smalltalk tags all objects with individual serial numbers. (Smalltalk is also notable for having two equality operators: "=" for separate-but-equal objects and "==" for selfsame objects.)

Always the Same

A big advantage of the serial-number approach to identity is that things stay the same even as they change. Identity doesn't depend on location or on any combination of attributes. Two bank accounts might have exactly the same balance, but they are different accounts because they have different account numbers. Within a single account the balance is likely to vary from day to day, but it remains the selfsame account.

This interplay of constancy and change is certainly a familiar feature of human life. My friend Dennis Flanagan has written that the molecules in most of the tissues of the human body have a residence half-life of less than two weeks. Clearly, then, I'm not the man I used to be—and yet I am. Indeed, it is when this process of continual molecular replacement ceases that "I" vanish.

In the semantics of programs, the unique identity of objects matters only when things *can* change. In a programming system without assignment operators or other ways of modifying existing values, the distinction between separate-but-equal things and the selfsame thing is of no consequence. If an object can never change after it is created, then the outcome of a computation will never depend on whether the program uses the original object or an exact copy.

For certain abstract kinds of objects, the whole concept of individual identity seems beside the point. In the equation $2x - 2 = x + 2$, should we think of the three 2's as being three separate-but-equal entities, or are they three expressions of a single archetype of 2-ness? It doesn't seem to matter. There is no way of telling one 2 from another. The same can be said of other abstractions, such as alphabetic characters or geometric points.

Even some elements of the physical world share this indifference to individuality. Electrons and other elementary particles seem to be utterly featureless; unlike snowflakes, no two are different. All electrons have exactly the same mass and electric charge, and they carry no serial numbers. They are a faceless multitude. No matter how long and hard we stare, there is no way to tell them apart. They are all separate but equal.

Or else maybe they are all the selfsame electron. In 1948 John Archibald Wheeler, in a telephone conversation with his student Richard Feynman, proposed the delightful hypothesis that there is just *one* electron in the universe. The single particle shuttles forward and backward in time, weaving a fabulously tangled "world line." At each point where the particle's world line crosses the space-time plane that we perceive as "now," it appears to us as an electron if it is moving forward in time and as a positron if it is going backward. The sum of all these appearances constructs the material universe. And that's why all electrons have the same mass and charge: because they are all the same electron, always equal to itself.

Bibliography

Baker, Henry G. 1993. Equal rights for functional objects or, the more things change, the more they are the same. *ACM OOPS Messenger* 4(4):2–27. Also available at ftp://ftp.netcom.com/pub/hb/hbaker/ObjectIdentity.html

Bird, Richard. 1976. *Programs and Machines: An Introduction to the Theory of Computation*. New York: John Wiley & Sons.

Flanagan, Dennis. 1988. *Flanagan's Version: A Spectator's Guide to Science on the Eve of the 21st Century*. New York: Alfred A. Knopf.

Kent, William. 1991. A rigorous model of object reference, identity, and existence. *Journal of Object-Oriented Programming* 4(3):28–36.

Khoshafian, Setrag N., and George P. Copeland. 1986. Object identity. *OOPSLA '86 Proceeding (Conference on Object-Oriented Programming Systems, Languages and Applications), Sigplan Notices* 21(11):406–416.

Minsky, Marvin. 1988. *The Society of Mind.* New York: Touchstone/Simon & Schuster.

Ohori, Atsushi. 1990. Representing object identity in a pure functional language. *Proceedings of the Third International Conference on Database Theory*, Paris, December 1990. Berlin, New York: Springer-Verlag.

Pacini, Giuliano, and Maria Simi. 1978. Testing equality in Lisp-like environments. *BIT* 18:334–341.

Peano, Giuseppe. 1889. The principles of arithmetic, presented by a new method. In *Selected Works of Giuseppe Peano*, translated and edited by Hubert C. Kennedy. Toronto, Canada: University of Toronto Press, 1973.

Russell, Bertrand. 1956. On denoting. In *Logic and Knowledge: Essays 1901–1950*. Macmillan, pp. 39–56.