How Computing Science created a new mathematical style

by

Edsger W. Dijkstra
University of Texas at Austin

To begin with, I would like to talk to you about equality. I don't know how you were introduced to the notion, I don't even remember how I was introduced to it myself. In our earliest memories, the notion was already there, and all of us must have appealed almost subconsciously to its obvious properties such as symmetry and transitivity, i.e., for all x, y, z

(0) $\quad x = y \;\equiv\; y = x \quad$, and

(1) $\quad x = y \;\wedge\; y = z \;\Rightarrow\; x = z \quad$,

respectively. But what status should we give to those mathematical facts we grew up with? Do we make them postulates or theorems?

I am perfectly willing to accept equality's symmetry as a postulate, in fact I prefer it that way. I prefer to view equality as defined on an unordered pair. (I similarly prefer products to be defined on bags of factors rather than on sequences of factors. In the last case, one of the first things one has to do is to point out that as far as the value of the product is concerned, the order in which the factors occur in the sequence is irrelevant. Why then order them in the first place?) The presence of (0) as rewrite rule merely reflects the absence of a canonical representation for the enumerated unordered pair which forces upon us the irrelevant choice between writing $x=y$ or $y=x$.

Equality's transitivity, however, I prefer to prove. Equality and function application are closely tied by Leibniz's Principle, i.e., the postulate that function application is equality-preserving: for all x, y, f of the appropriate types

(3) $\quad x = y \;\Rightarrow\; f.x = f.y$.

It is understood that expressions can be viewed as functions applied to their sub-expressions, so that

$$a = b \;\Rightarrow\; (a+3)^2 = (b+3)^2$$

is an instantiation of (3) – without invoking the whole mechanism of the λ-calculus —.

Leibniz's Principle (3) can be rephrased equivalently by stating that for all x, y, z, f of the appropriate types

(4) $\quad x = y \wedge f.y = z \Rightarrow f.x = z$

(a kind of reformulation we shall return to later). Instantiation of (4) with for f the identity function yields (1). In other words: the transitivity of equality is a special case of Leibniz' Principle.

The links between equality and function application are very strong: function application preserves equality, conversely: apart from the constant relations true and false, equality is the only relation that is preserved by function application. In the many formal proofs I designed, Leibniz' Principle is invoked all the time and I believe that the only use I make of the notions of equality and of function application is in appeals to Leibniz's Principle.

It is a style of doing mathematics that is rather different from what was deemed acceptable only two decades ago.

In his book "Graphs and Hypergraphs" (1970), Claude Berge introduces the following graph

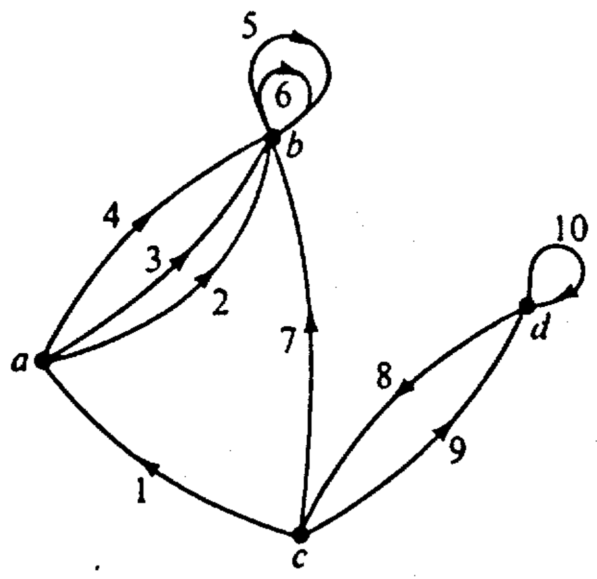

Fig. 1.1. A 3-graph of order 4

and then is willing to code the fact that arc. 9 goes from c to d as

$$9 = (c, d) \quad ;$$

this notational convention is asking for trouble since it allows us to deduce from the picture $2 = (a, b)$ and $3 = (a, b)$, from which $2 = 3$ follows!

I think we all agree that Berge's convention is a misuse of Recorde's equality sign. We may have different opinions as to how serious this misuse is. The classical defence of a notation like Berge's is that everybody knows what is meant and that no one worth his keep will derive such obvious nonsense as $2 = 3$. Fair enough, but beside the point. A fundamental and therefore potentially much more serious objection is that such notational conventions create environments hostile to derivations by formula manipulation. We shall return to the possible weight of this objection later, staying a little bit longer with the equality.

It may surprise us at first sight that a relation so simple and familiar as equality has created all sorts of problems, but it has created them precisely because of its familiarity. Since antiquity, mathematicians have never been sure how to indicate it, if it had to be indicated at all: sometimes it was done verbally, sometimes by an acronym of such verbal statement, sometimes equal expressions on a page were connected by a line, often there was no more than juxtaposition. The more interesting history of the equality starts 4 centuries ago, in 1557, when Robert Recorde designed the equality symbol as we know it today. It had to wait for more than a century until Leibniz (1646-1710) formulated its crucial property. It had to wait 3 centuries, viz. until 1854, before it acquired in the hands of George Boole the full status of an infix operator that makes $A = B$ an expression which may take on a value.

And the speed with which the world can absorb improvements is limited. A century later, the boolean domain was still a second-class citizen of the mathematical world. My comment on Berge's convention is the following: in view of the fact that it has taken the equality sign more than 400 years to occupy in mathematics the position it deserves, we should consider it a mathematical treasure to be protected from polution.

Let me now illustrate what we can do with the equality and its transitivity when its operands are boolean. In that case it is not unusual —though regrettable— to pronounce it as "if and only if", as in the following

<u>Theorem</u> $(C,<)$ is well-founded if and only if mathematical induction over C with respect to $<$ is a valid proof technique. (End of Theorem.)

The pronunciation "if and only if" suggests that such a theorem has to be proved by mutual implication, i.e., by demonstrating that either side follows from the other. That such is not the case is shown by the following calculation; for the first transformation one needs to know that well-foundedness means that each non-empty subset contains a minimal element, for the next two predicate calculus suffices, and the last one appeals to the definition of mathematical induction. In the following calculation

- dummies x and y are elements of C
- dummy S is a subset of C
- dummy P is a predicate on C .

$(C,<)$ is well-founded

$=$ {definition of well-foundedness}

$(\forall S:: (\exists x:: x\in S) \Rightarrow (\exists x:: x\in S \wedge (\forall y: y<x: y\notin S)))$

$=$ {contrapositive and Laws of de Morgan}

$(\forall S:: (\forall x:: x\notin S) \Leftarrow (\forall x:: x\notin S \vee (\exists y: y<x: y\in S)))$

$=$ {transforming the dummy: $S=\{x|\neg P.x\}, P.x \equiv x\notin S$}

$(\forall P:: (\forall x:: P.x) \Leftarrow (\forall x:: P.x \vee (\exists y: y<x: \neg P.y)))$

$=$ {definition of proof via mathematical induction}

mathematical induction over $(C,<)$ is valid .

This is not the place to discuss the above proof character by character. Its combination of brevity and completeness is unsurpassed. (Less than 40 years ago, the proof of this theorem has required two lectures of 45 minutes each: one for each direction of the implication.) Moreover, its design requires no more than experience in formula manipulation.

As next example we show how taking the maximum distributes over taking the minimum. Denoting the maximum of x and y by $x{\uparrow}y$ and their minimum by $x{\downarrow}y$, the theorem to be proved is that for all x, y, z

(5) $\quad x{\uparrow}(y{\downarrow}z) = (x{\uparrow}y){\downarrow}(x{\uparrow}z)$.

We use for the definition of $\uparrow$ and $\downarrow$ that for all a, b, w

(6) $\quad w \geq a{\uparrow}b \equiv w \geq a \wedge w \geq b$

(7) $\quad w \geq a{\downarrow}b \equiv w \geq a \vee w \geq b$,

and in view of the above we prove equality by using

(8) $\quad a = b \equiv (\forall w :: w \geq a \equiv w \geq b)$.

To demonstrate (5), we observe for any x, y, z, w

$w \geq x{\uparrow}(y{\downarrow}z)$

= $\quad$ { (6) with $a, b := x, (y{\downarrow}z)$ }

$w \geq x \wedge w \geq y{\downarrow}z$

= $\quad$ { (7) with $a, b := y, z$ }

$w \geq x \wedge (w \geq y \vee w \geq z)$

= $\quad$ {predicate calculus, in particular: $\wedge$ distributes over $\vee$ }

$(w \geq x \wedge w \geq y) \vee (w \geq x \wedge w \geq z)$

= $\quad$ { (6) with $a, b := x, y$ and $a, b := x, z$ }

$w \geq x{\uparrow}y \vee w \geq x{\uparrow}z$

$=\quad \{(7) \text{ with } a,b := (x\uparrow y),(x\uparrow z)\}$
$\quad w \geq (x\uparrow y)\downarrow(x\uparrow z)$,

which, in view of (8), proves (5) .

A few things should be remarked about this argument, besides the fact that it avoids all case analyses and beyond definitions (6) and (7) does not use a single property of the relation $\leq$. Like the argument about mathematical induction it makes extensive use of the notion of equality of boolean expressions. Like reformulation (4) of Leibniz's Principle, our formulae (6), (7), and (8) contain a dummy more than perhaps expected. Such "extra" dummies give us more manipulative freedom, and they are often the key to surprisingly crisp arguments. (It helps to know besides (8)

(9) $\quad a \geq b \equiv (\forall w :: w \geq a \Rightarrow w \geq b)$.)

To close this section, let me tell you an experiment all of you can take when you are home again. Ask the average -and even not so average- mathematician to prove or refute for real a,b

$a+b \geq \max(a,b)$ if and only if $a \geq 0 \wedge b \geq 0$

and you will be surprised by the inefficiency of his argument compared to

$$\begin{array}{ll} & a+b \geq a \uparrow b \\ = & \quad \{ (6) \text{ with } w := a+b \} \\ & a+b \geq a \ \wedge \ a+b \geq b \\ = & \quad \{\text{arithmetic}\} \\ & b \geq 0 \ \wedge \ a \geq 0 \end{array}$$.

The explanation for the usual inefficiency of the argument is that it starts with a definition of max like

$$\max(a,b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases} ,$$

a definition which is not nice to manipulate, and therefore invites case analysis. (As my Eindhoven colleague W.H.J. Feijen, to whom I owe this example, writes in reference to the above definition: "The reader should try to use it in proving the associativity of max.".)

* * *

I hope that the above little examples give you some idea of the calculational style of reasoning that computing science developed and adopted, but before we try to estimate the wider significance of that development, we had better understand why computing science chose that path.

There is a combination of reasons, ranging from need to opportunity.

The need emerged because computers are so unforgiving: they execute programs as written, and not programs as dreamt. Getting our software correct turned out to be surprisingly hard. Closer scrutiny revealed that it was not just a matter of inaccuracy, of typing errors that better proofreading would have caught. The situation was much more serious: the programming challenge confronted us with a type of complexity we did not know how to cope with. If the amount of reasoning that a program requires for its justification explodes with the program length, only the tiniest programs will be free of bugs!

Computing scientists quickly discovered that the amount of reasoning required could critically depend on nature and structure of the interfaces between modules; the competent system designer therefore designs his interfaces with the utmost care. In mathematics, definitions and theorems are interfaces; they should be chosen with similar care, but are they? Not always. (Hardy and Wright: "Every positive integer, except 1, is a product of primes"; Harold M. Stark:

"If n is an integer greater than 1, then either n is prime or n is a finite product of primes.". These examples —which I owe to A.J.M. van Gasteren— both reject the empty product, the last one also rejects the product with a single factor.)

The other technique used to reduce the amount of reasoning needed was not to try to prove the correctness of existing programs, but to design correctness proof and program together. More precisely: one designs the correctness proof and derives the program from that proof. (That then the correctness proof tends to be simpler than the correctness proof for an intuitively conceived program is not surprising. What is surprising is that such proof-driven programming often leads to the more efficient program.)

Next I ask you to realize that, by virtue of the possibility of mechanical interpretation, each programming language represents a formal system (of sorts). That means that programs can —and in a way have to— be viewed as formulae. In the previous paragraph I spoke about "deriving programs from proofs"; we now see that that means that formulae have to be derived from the proofs. Without further detail, the above should explain why formula manipulation is so often the computing scientist's preferred way of proving theorems.

This preference is also understandable in the light of the rest of his professional experience. If we believe Halmos, the manipulation of uninterpreted formulae plays a very minor rôle in the life of the classical mathematician: he has not been trained to do it, he does not like it, and he is not sure that he considers calculational proofs really convincing. The manipulation of uninterpreted formulae is, however, a most familiar operation for the computing scientist: it is the one and only operation computers are very good at.

Moreover, the computing scientist is spared an emotional barrier. Almost all formalisms used daily by the classical mathematician are at least ambiguous. But that does not hurt the classical mathematician because he does nothing with a formula without interpreting it and part of his professional competence consists in subconsciously rejecting all unintended interpretations. (If you show him

$$\sin(\alpha+\beta) \quad \text{versus} \quad \sin(s+i+n) \quad ,$$

it depends on his sense of humour whether he is amused.) The manipulation of uninterpreted formulae requires unambiguous formalisms; I have observed more than once that the classical mathematician experiences such disambiguations as irritating priggery. The hurdle is understandable, but the computing scientist has taken it a long time ago.

* * *

How relevant is the above mathematical experience of the computing scientists for mathematics in general? Rather relevant I think, and I do so for mainly two reasons.

The one reason is that the calculational proofs are almost always more effective than all informal alternatives, the other reason is that the design of calculational proofs seems much more teachable than the elusive art of discovering an informal proof.

Let me elaborate on the first reason first. When I called calculational proofs more "effective", I meant that they tend to be shorter, more explicit, and so complete that they can be read and verified without pencil and paper. To find the calculational proofs we had to design so effective was a very pleasant surprise, and it was a surprise indeed, because we too had been subjected, over and over again, to the rumour that formal methods were too clumsy to be used in practice. My conclusion is that that rumour is wrong.

I think I know how the misconception of this rumour emerged: the initial experiences with formality were very discourageing indeed. These unfortunate experiences were due to our lack of manageable formalisms geared to our manipulative needs and to our lack in expertise in using formalisms. Our disappointing experiences with formality should not be interpreted as something being wrong with formality; the experiences were disappointing because we were incompetent amateurs. But this has been changed by our exposure to computing because experience gained there can indeed be carried over to mathematics in general. Let me give you two examples.

In order to prove that a compiler correctly parses the program texts it processes, one needs a formal grammar for the syntax of the programming language in question. To formal language theory we owe the distinction between grammars that are "context-free" and those that are not. Next we observe that the manipulating mathematician does not manipulate just strings of symbols but manipulates formulae parsed according to the underlying grammar. Since the parsing problem is simpler for context-free grammars, the moral is simple: choose for one's formalisms context-free grammars.

The other example is the notion of abstract data types —stacks, trees, permutations, bags, configurations, what have you—. Identifying an appropriate data type and some operations on it is often essential in preventing a program from becoming totally unwieldy. In exactly the same way calculational proofs of the desired brevity and clarity can often be obtained by the use of a specifically designed, special-purpose calculus. I said "exactly the same way" because I see no difference between the virtues of a data type in a program and the virtues of a calculus in a proof.

That concludes what I wanted to say about the effectiveness of calculational proofs. Let me now turn to the second reason why I think that our experience with calculational proofs is relevant, viz. the teachability of their design.

Compared to the stern discipline of formality, informal mathematics is a free-for-all. Because of the very strict requirements a formal proof has to meet, the design of a formal proof is widely felt to be a more demanding task than the design of an informal proof.

This feeling, however, is not justified: formal proofs are in a very precise way easier to design than informal ones. Because the requirements of formality restrict our freedom, we have fewer options for trials that turn into errors: our "search space" has been reduced, and that helps. Furthermore, accepting the requirements of formality means reducing the bandwidth of what we have to write down, a bandwidth reduction that allows us to adopt for our calculations a strict format. The advantages of such a strict format are considerable. Firstly, it makes proofs of different theorems more similar, and also the tasks of designing those proofs: problems from different areas can easily lead to similar challenges in formula manipulation, thus increasing the probability that the proof design can be guided by the same heuristics. Secondly, the strict format makes that many a design decision can be taken on syntactic grounds alone.

Let me give you an example. I know that it may be too simple to be impressive. It has the advantage that it can be related to the material mentioned before.

Let function f distribute over max, i.e. for all a, b

(10) $\quad f.(a \uparrow b) = f.a \uparrow f.b \quad ;$

show that f is monotonic, i.e. for all x, y

(11) $\quad x \geq y \;\Rightarrow\; f.x \geq f.y \quad .$

The proof uses how to express $\geq$ in terms of $\uparrow$, i.e., for all a, b

(12) $\quad a \geq b \;\equiv\; a = a \uparrow b$

$$
\begin{array}{ll}
 & f.x \geq f.y \\
= & \quad \{ (12) \text{ with } a,b := f.x, f.y \} \\
 & f.x = f.x \uparrow f.y \\
= & \quad \{ (10) \text{ with } a,b := x,y \} \\
 & f.x = f.(x \uparrow y) \\
\Leftarrow & \quad \{ \text{Leibniz} \} \\
 & x = x \uparrow y \\
= & \quad \{ (12) \text{ with } a,b := x,y \} \\
 & x \geq y \quad .
\end{array}
$$

It is worth noticing that this proof is forced. Look at our proof obligation (11). Going from consequent to antecedent, two f applications have to be eliminated, so we <u>have</u> to apply the Principle of Leibniz, because that is the <u>only</u> one that can achieve that for us. The structure of the proof is now dictated: (11)'s consequent has to be massaged into the equality of two f-values, and after the elimination of the two f-applications, the equality sign has to be eliminated. Moreover, we have to use (10), which mentions $\uparrow$, which does not occur in (11). So, as $=$, $\uparrow$ has to be introduced and eliminated again. This raises the question: how can we exchange an $\geq$ for an $\uparrow$ and and $=$? Formula (12) gives the answer; if we don't know it, we can decide most of its structure beforehand: for the sake of applicability of (10), the right-hand side of (12) has to contain $a\uparrow b$. The proof of (12) is left as an exercise to the reader; besides predicate calculus and (6), he needs that $\geq$ is reflexive and antisymmetric, i.e. for all a, b

$$a=b \equiv a\geq b \wedge b\geq a \quad .$$

* * *

When your Mathematische Gesellschaft in Hamburg was founded, your countryman Gottfried Wilhelm Leibniz was in the prime of his life. For centuries, his dream that calculation should provide an alternative to human reasoning remained a dream. I hope to have shown you how the existence of automatic computers and the challenge of using them has changed this:

simplifications that Leibniz had in mind are now within reach. The moral of my story is that if we treat our formalisms with the care and respect that we pay to our other subtle artefacts, our care and respect will be more than rewarded. Calculemus!

I thank you for your attention.

Austin, 4 March 1990

prof. dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA