Mathematical Induction and Computing Science

Edsger W. Dijkstra
Burroughs / Eindhoven University of Technology

I would like to use the opportunity of addressing a general mathematical audience to give that audience some feeling for computing science's mathematical component. The reason is simple. Whenever a new area of human endeavour has been shown to be amenable to mathematical treatment, this has always had its impact on mathematics at large. And when the new demands made on our powers of mathematical reasoning are somehow drastically different from what we were used to, that impact can even be expected to be quite profound.

To avoid misunderstanding, a few introductory remarks first. Each application area of mathematical thought requires for its effective treatment its own mathematical style, its own formalisms and notations, its own peculiar balance between what is explicitly stated and what is implicitly understood and between the application of formal techniques and of common sense. For a new area, neither the appropriate symbolisms nor the proper balances are obvious; in practice, consensus is only reached after many, many experiments and is as such a product of time and hard work.

Mathematicians that mainly work in by now well-established areas and were never involved in such struggles tend to take the availability of an adequate style for granted and to equate —in their world not without justification— "convenient" with "conventional". In this respect, today's computing scientist could not be more different: he finds that all sorts of mathematical habits, developed in the past for other purposes, are inadequate for his purpose and is forced to the recognition that time-honoured conventions of the mathematical world are often in a very objective sense highly inconvenient. As a result it is not unusual that the computing scientist regards his colleague in more traditional branches of mathematics as conservative beyond redemption, while the latter considers his colleague in computing a dangerous and barbarian radical. Such mutual ill feelings are, however, no more than the predictable outcome of underestimating the differences between the challenges the mathematical computing scientist faces and the challenges we faced before the advent of automatic computers, when, for lack of machines, programming was no problem at all. Let me, therefore, sketch the peculiar nature of the challenge faced by the mathematical computing scientist.

Programs are written in a programming language, which by the very fact of its mechanical interpretation represents a formal system of some sort. Each program text is a formula. It represents, to be precise, a predicate on the Cartesian product of the so-called initial state space and the final state space. Formally deriving a program from its specifications or proving the equivalence of two programs is no more and no less than a particular form of predicate calculus. I said "particular form of predicate calculus" because a predicate expression has to satisfy some criteria of constructiveness if it is to allow the interpretation of automatically executable code.

With the interpretation of a program text as a predicate expression we are at least conceptually on more or less familiar grounds. I said "more or less familiar" since it is my impression that the average mathematician knows of the existence of the classical predicate calculus but rarely uses it. This is in sharp contrast with the computing scientist, for whom the predicate calculus has become an indispensable tool for his daily reasoning. I said that the grounds were "at least conceptually" familiar: they are familiar to the extent that the average mathematician feels quite at home with the idea of manipulating formal expressions provided those expressions are not too big, say up to a few lines. But programs of a few hundred lines are not unusual at all. It is this scaling up that profoundly changes the rules of the game: any child can control a bicycle but you need a professional pilot to fly a supersonic plane. For cyclist and pilot, the laws of mechanics are the same, yet the conventional cyclist's wisdom is of no great relevance in the cockpit.

* * *

I have chosen to talk about mathematical induction since that is the indispensable tool for arguing about recursively defined structures and recursive definitions pervade all of computing science. It already starts with the definition of the syntax of our programming languages. The usual form is BNF ("Backus-Naur-Form"). After the definition of digits

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

the syntax of unsigned integers is given by

```
<unsigned integer> ::= <digit>
      |<unsigned integer><digit>
```

— in words: an unsigned integer is a single digit or an unsigned integer followed by a digit — and after the definition of what letters are, identifiers are defined by

```
  <identifier> ::= <letter>
      | <identifier> <letter> | <identifier><digit>    .
```

An example of a more complicated recursive definition would be the following definition of expressions

```
  <addop> ::= + | -

  <expression> ::= <product> | <addop> <product>
       | <expression> <addop> <product>

  <multop> ::= * | /

  <product> ::= <factor>
      | <product> <multop> <factor>

  <factor> ::= <unsigned integer> | <identifier>
      | (<expression>)          .
```

The way in which in its last line parentheses are introduced illustrates the mechanism of the recursive definition in its full glory.

* * *

The most common formulation of mathematical induction is: from

(the base)	$P\,0$
(the step)	$(\underline{A}\, n: n \geq 0: P n \Rightarrow P(n+1))$
follows	$(\underline{A}\, n: n \geq 0: P n)$.

Other formulations of the step are

$$(\underline{A}n: n>0: P(n-1) \Rightarrow Pn)$$

and

$$(\underline{A}n: n>0: \neg P(n-1) \vee Pn) \quad .$$

Next we have the "course-of-values induction" —usually derived from the former formulation— : from
(the base) $P0$
(the step) $(\underline{A}n: n>0: (\underline{A}i: 0\leq i<n: Pi) \Rightarrow Pn)$
follows $(\underline{A}n: n\geq 0: Pn)$,

with "infinite regress" as notational variant for the step:

$$(\underline{A}n: n>0: \neg Pn \Rightarrow (\underline{E}i: 0\leq i<n: \neg Pi)) .$$

A more modern formulation is

$$(\underline{A}n: n\geq 0: Pn) =$$
$$(\underline{A}n: n\geq 0: Pn \vee (\underline{E}i: 0\leq i<n: \neg Pi))$$

Note that

0) by expressing the implications as a disjunction, we made the difference between "course-of-values induction" and "infinite regress" —a notational artefact!— disappear;
1) the base has been subsumed in the step;
2) the whole statement is not one of implication but of <u>equality</u> of predicates.

But let us now depart from the natural numbers. Let us introduce the following nomenclature:

$(U, <)$ is a partially ordered set
x, y are elements from U
S is a subset from U
P is a total predicate on U, where S and P are coupled by

$$Px = \neg(x \in S) \quad \text{or} \quad S = \{x \mid \neg Px\} \quad ,$$

where we have for each pair S, P :

$$(S = \emptyset) = (\underline{A}\, y : y \in U : Py) \quad .$$

"y is a minimal element of S" means

$$(y \in S) \wedge (\underline{A}\, x : x < y : \neg(x \in S)) \quad .$$

"U is well-founded" means that each non-empty subset of U contains a minimal element.

"U is well-founded"
$= (\underline{A}\, S :: (S \neq \emptyset) =$
$\qquad (\underline{E}\, y : y \in U : y \in S \wedge (\underline{A}\, x : x < y : \neg(x \in S))))$
$= (\underline{A}\, S :: (S = \emptyset) =$
$\qquad (\underline{A}\, y : y \in U : \neg(y \in S) \vee (\underline{E}\, x : x < y : x \in S)))$
$= (\underline{A}\, P :: (\underline{A}\, y : y \in U : Py) =$
$\qquad (\underline{A}\, y : y \in U : Py \vee (\underline{E}\, x : x < y : \neg Px)))$.

From the above we conclude that well-foundedness is exactly the same thing as validity of a proof by mathematical induction.

A "decreasing chain" is a sequence $x_0, x_1, x_2, \ldots$ of elements such that $(\underline{A}\, i, j : 0 \leq i < j : x_j < x_i)$.

"U is well-founded" = "all decreasing chains in U are of finite length" .

Proof. With the predicate DCF defined by

DCF y = "all decreasing chains of which y is the first element are of finite length",

we have to prove

"U is well-founded" = $(\underline{A} y: y \in U: \text{DCF } y)$

Assume the left-hand side true; then the right-hand side equals

$$(\underline{A} y: y \in U: \text{DCF } y \lor (\underline{E} x: x < y: \neg \text{DCF } x)) \quad ,$$

which is evidently true.

Assume the left-hand side false; then there exists a non-empty S such that it has no minimal element, i.e.

$$(\underline{A} y: y \in S: (\underline{E} x: x < y: x \in S))$$

from which the falsity of the right-hand side is evident. (End of Proof.)

The above theorem shows the extremely close connection between well-foundedness and proofs of termination of a program. Let U be the state space of the computations, and let $x < y$ mean that x is a possible successor state of state y . Each computation then corresponds to a decreasing chain and "all computations terminate" means that all those decreasing chains are finite, i.e.

that the successor relation turns U into a well-founded set. Note that we have used equalities all over the place: terminating computations, well-founded sets, and the validity of inductive arguments are three faces of the <u>same</u> coin.

As a result, computing science has a vivid interest in well-founded sets and their construction. Lexicographic pairing is a beloved construction. If U and V - with general elements u and v respectively - are well-founded, their Cartesian product $U \times V$ is also well-founded under the ordering defined by

$$(u_0\ v_0) < (u_1\ v_1) = u_0 < u_1\ \vee ((u_0 = u_1) \wedge\ v_0 < v_1) \quad .$$

Lexicographic pairing has the further advantage of being associative. It is total iff the component orderings are total.

Proving properties of all sentences of a programming language is done by "induction over the syntax". This is most elegantly done by introducing a partial order on strings: $x < y$ means that string x occurs as syntactical component of string y ; this partial order is, however, easily embedded in a total order, viz. string length. But sometimes we have to carry out inductive arguments over essentially only partially ordered sets, e.g. when the order relation is "earlier in time", but we are dealing with a network in which events in distinct components are essentially unordered for lack of communication facilities between such components, and the lacking possibility of synchronization is at the heart of the problem to be solved.

Nuenen, 4 April 1982