Formality Works

Rob R. Hoogerwoord [1]

department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

Formal calculations can contribute successfully to a better understanding of the structure of a proof, both in programming and in "classical" mathematics. This is illustrated by means of a few examples.

Key words: calculational mathematics, transitive closure, program derivation, distributed computing

I always think very hard before I say something stupid.
Loesje

1 Introduction

In secondary school, my mathematics teacher used to react to too complicated solutions with a Dutch proverb: "That is shooting a sparrow with a cannon", to which he often added with an ironic smile: "of course, the poor little creature is dead allright". Thus, by introducing me not only to mathematical proof and rigour but also to the importance of simplicity in reasoning, he has laid the foundation for the development of my professional attitude. To this attitude Edsger W. Dijkstra has contributed even more, by his rigour, his quest for simplicity and elegance, and, most important of all, his insistence on independent thinking and not taking anything for granted.

Technically speaking, the most important things Dijkstra has taught me are the principle of *Separation of Concerns* and the insight that programs and their proofs must be developed simultaneously; this is also called *Correctness by Design* and is based on the understanding that the specification, and the obligation to prove that the program will meet it, is all we have got for a start, so we had better use it: we really have no choice here. Both Separation

[1] E-mail: R.R.Hoogerwoord@tue.nl

of Concerns and Correctness by Design share the sad fate of being not so well-known as they deserve and, in some circles, of even being explicitly rejected.

In particular, in a mathematical text the *use* of a property and its *proof* are separable concerns: we can use a property without supplying proof for it; of course, the *validity* of such use requires (the existence of at least one) proof, but we may supply this proof "elsewhere", inside or outside the same text. (Also, only *one* proof is needed, independent of the *number* of uses!) In this paper, for example, I use several properties without proof, for the sake of brevity and because I do not need these proofs to convey my message. Their omission does not imply that they would not be of interest – on the contrary! –: I only have separated my concerns.

2 Calculemus

An as yet insufficiently recognized but undeniable development is the emergence of a more calculational style of problem solving, applicable to the construction of both programs and proofs. In this style symbols and formulae play a much more important role than in traditional mathematics. The advantages of such a calculational style are:

- the shape of the formulae provides heuristic guidance;
- assumptions and other properties used are made explicit,
- and so are design decisions;
- thus, the mathematical structure of the design is clarified.

Here is a simple example. We wish to prove that $\sqrt{p}$ is irrational, for a given and, for the time being fixed, prime number p. To this end, we introduce a function f, with the following (desired) properties:

$$\begin{array}{lll} (0) & f{\cdot}p & = \ 1 \\ (1) & f{\cdot}(x{*}y) & = \ f{\cdot}x + f{\cdot}y \quad , \text{ for positive natural } x, y \ . \end{array}$$

As to the valid but separable concern of f's existence: (0) and (1) are satisfied if $f{\cdot}x$ is the number of times x is divisible by p, for all positive natural x (but this is only one possible definition). Now we perform the following calculation – in Wim Feijen's format and where x, y are positive naturals –:

$$\begin{array}{ll} & \text{“}\sqrt{p} \text{ is rational”} \\ \equiv & \quad \{ \text{ definition of “rational” } \} \\ & (\exists x, y :: \sqrt{p} = x/y \,) \\ \equiv & \quad \{ \text{ definitions of } \sqrt{\ } \text{ and } / \ \} \end{array}$$

$$
\begin{array}{ll}
 & (\exists x, y :: p{*}y^2 = x^2) \\
\Rightarrow & \quad \{ \text{ Leibniz, to introduce } f \text{ } \} \\
 & (\exists x, y :: f{\cdot}(p{*}y^2) = f{\cdot}(x^2)) \\
\equiv & \quad \{ \text{ properties of } f \text{ } \} \\
 & (\exists x, y :: 1{+}2{*}f{\cdot}y = 2{*}f{\cdot}x) \\
\equiv & \quad \{ \text{ odd numbers differ from even ones } \} \\
 & \mathsf{false} \quad ,
\end{array}
$$

from which we conclude that $\sqrt{p}$ is irrational. In this proof we have not needed to assume that x and y have no common divisors – in contrast to other renderings of the "same" proof –, nor have we used mathematical induction explicitly: the latter is captured by the properties of f.

In the above proof I seem to have used nowhere that p is prime: so, the theorem is also valid for all other numbers? Of course not: that p is prime plays a role in the definition of f, because properties (0) and (1) are not satisfiable for all numbers p. Thus, calculational reasoning contributes to – and also is only possible with! – a good modularisation of the proof.

The crux of the above proof is the property: $(\forall x, y :: 1{+}2{*}y \neq 2{*}x)$, and this immediately points to a generalisation, because it is an instance of the following, more general property:

$$(\forall x, y :: r{+}n{*}y \neq n{*}x) \quad , \text{ for all } \; r, n : \; 0{<}r{<}n \quad .$$

Now it is no big step anymore – we leave this as an exercise – to construct the following, more general theorem, where p now ranges over all primes and f_p denotes the function, as above, associated with prime p:

For all positive natural n and z:

"$\sqrt[n]{z}$ is rational" $\quad \equiv \quad (\forall p :: f_p{\cdot}z \bmod n = 0)$.

The theorem is an *equivalence* now: it states the exact – necessary and sufficient – condition for rationality of any root. To *apply* this theorem and to evaluate this condition we need a full definition of f, but to *prove* the theorem we only need (0) and (1): that is Separation of Concerns too.

3 Transitive closure and graph connectivity

The usefulness of a mathematical notion in calculational reasoning also depends on the *shape* of its definition. This becomes particularly relevant when

the same notion admits of different but equivalent definitions. In such a case it may even pay to formulate several such definitions, and to establish their equivalence, so that we can use the most appropriate one whenever needed.

We illustrate this by means of the *transitive closure* and some of its applications. For a binary relation $\sim$ on some (anonymous) universe, we denote its transitive closure by $\overset{+}{\sim}$. One way to define $\overset{+}{\sim}$ is a recursive one.

definition 0:

$$(\forall x, z :: \; x \overset{+}{\sim} z \;\equiv\; x \sim z \vee (\exists y :: x \sim y \wedge y \overset{+}{\sim} z)\,)$$

□

This definition already allows several variations, which we shall not further pursue here, obtained by different placements of $\sim$ and $\overset{+}{\sim}$ in the right-hand side of the definition. An equivalent but entirely different way to define $\overset{+}{\sim}$ is more implicit, by stating its relevant properties right away. In words, this defines $\overset{+}{\sim}$ as *the strongest* of all transitive relations that are as weak as $\sim$.

definition 1:

$$(\forall x, z :: x \sim z \Rightarrow x \overset{+}{\sim} z) \quad \text{and} \quad \overset{+}{\sim} \text{ is transitive ,}$$

and every transitive relation $<$ satisfies:

$$(\forall x, z :: x \sim z \Rightarrow x < z) \quad \Rightarrow \quad (\forall x, z :: x \overset{+}{\sim} z \Rightarrow x < z) \quad .$$

□

We now consider an undirected graph, which is a set of so-called *nodes* and a symmetric binary relation $\sim$ on that set of nodes. In what follows, dummies x, y, z range over the nodes. Nodes related by $\sim$ are also called *neighbours*.

Traditionally, two nodes x and z are called *connected* if a *path* exists from x to z, which is a sequence $y_0, y_1, \cdots, y_n$, for some $n : 1 \leq n$, with:

$$x = y_0 \;\wedge\; (\forall i : 0 \leq i < n : y_i \sim y_{i+1}) \;\wedge\; y_n = z \quad .$$

This is an awkward definition, because of its abundant nomenclature; when we discuss connectivity, we are only interested in the *existence* of paths, not in their identity; so, naming paths, their lengths, and their elements is overdone.

Once we are aware of this, we can define connectivity recursively: x and z are connected either if they are neighbours or if x has a neighbour y and y and z are connected. Now this has precisely the shape of definition 0 ! Apparently, connectivity is just the transitive closure of the neighbour relation: x and z are connected means $x \overset{+}{\sim} z$.

The whole graph is called connected if every two nodes are connected; hence,

graph connectivity can be defined concisely by:

$$(\forall x, z : x \neq z : x \overset{+}{\sim} z) \quad .$$

Having introduced the transitive closure, we can exploit all of its general properties, for example, that it also satisfies definition 1. With f for some function on the nodes of the graph – one may think of a node *colouring* –, the binary relation defined by $f_x = f_z$ is transitive, and thus we obtain – directly from definition 1 –:

$$(2) \qquad (\forall x, z : x \sim z : f_x = f_z) \;\Rightarrow\; (\forall x, z : x \overset{+}{\sim} z : f_x = f_z) \quad .$$

In words: if every two neighbours have the same colour then every two connected nodes have the same colour. Hence, in a connected graph all nodes have the same colour if every two neighbours have the same colour. Conversely, if this is true for all possible node colourings, then the graph is connected: this may even be taken as an alternative definition of connectedness.

In the same vein, for some boolean function c on the nodes, the binary relation defined by $c_x \Leftarrow c_z$ is transitive, and thus we obtain:

$$(3) \qquad (\forall x, z : x \sim z : c_x \Leftarrow c_z) \;\Rightarrow\; (\forall x, z : x \overset{+}{\sim} z : c_x \Leftarrow c_z) \quad .$$

4 Distributed algorithms

Properties like (2) and (3) allow *global* conclusions to be drawn from *local* properties: their left-hand sides are about neighbours only, whereas their right-hand sides are about all connected nodes. I have applied this technique succesfully to a formal development of a distributed algorithm [1]; here we present a summary of the main mathematics involved in that development.

A class of elementary distributed algorithms are the *wave algorithms*. In terms of node colourings, such an algorithm boils down to: initially, all nodes have the same colour *white*, say, and the computation is initiated by one node, the *root*, by changing its colour to *red*, say. Every other white node becomes red only after at least one of its neighbours has become red first.

Formally, the state of node x can be represented by a boolean variable c_x, with interpretation:

$$c_x \;\equiv\; \text{“}x \text{ is red”} \quad .$$

The change of colour of any node x then amounts to $c_x := \mathsf{true}$. Except for the root, this is guarded by the condition $(\exists y : x \sim y : c_y)$.

As a *progress requirement* we must prove that the computation terminates, with all nodes red; because of the guards this is not entirely trivial. This requirement can be formulated as follows, expressing that either the desired final state has been reached, or at least one white node is allowed to become red – where R is the root node –:

$$(\forall x :: c_x) \;\vee\; \neg c_R \;\vee\; (\exists x :: \neg c_x \wedge (\exists y : x \sim y : c_y)) \quad .$$

We prove this by taking the negation of the third disjunct as starting point:

$$\begin{array}{ll}
 & \neg\,(\exists x :: \neg c_x \wedge (\exists y : x \sim y : c_y)) \\
\equiv & \quad \{\ \wedge \text{ over } \exists\,;\ \text{unnesting dummies; mainly De Morgan}\ \} \\
 & (\forall x, y : x \sim y :\ c_x \Leftarrow c_y) \\
\Rightarrow & \quad \{\ \text{transitive closure (see (3))}\ \} \\
 & (\forall x, y : x \stackrel{+}{\sim} y :\ c_x \Leftarrow c_y) \\
\Rightarrow & \quad \{\ \text{the graph is connected}\ \} \\
 & (\forall x, y ::\ c_x \Leftarrow c_y) \\
\Rightarrow & \quad \{\ \text{instantiation } y := R\ \} \\
 & (\forall x ::\ c_x \Leftarrow c_R) \\
\equiv & \quad \{\ \text{predicate calculus}\ \} \\
 & (\forall x ::\ c_x) \;\vee\; \neg c_R \quad .
\end{array}$$

This derivation shows explicitly that connectivity of the graph is *used* to prove progress. (That we also *need* it can be shown by counterexample.)

* * *

Suppose the root must *detect* that all nodes have become red. This is known as *termination detection* and it is a typical example of the need to establish a global property locally. For this purpose, a boolean variable b_x per node x is introduced, with as local invariant – that is, within node x –:

$$b_x \;\Rightarrow\; c_x \quad .$$

"All nodes red" now follows from $(\forall x :: b_x)$; to enable the root to establish this global property on local grounds, it would be nice if we would have:

$$(\forall x :: b_x \Leftarrow b_R) \quad ,$$

because then b_R alone would suffice to establish $(\forall x :: b_x)$.

To achieve this in a manageable way, a directed subgraph $\rightarrow$, with $\stackrel{+}{\rightarrow}$ for its transitive closure, of the whole graph is introduced, and we calculate:

$$
\begin{array}{ll}
 & (\forall x :: b_x \Leftarrow b_R) \\
\equiv & \quad \{ \text{ assume } (\forall x : x \neq R : x \stackrel{+}{\rightarrow} R) \ \} \\
 & (\forall x : x \stackrel{+}{\rightarrow} R : b_x \Leftarrow b_R) \\
\Leftarrow & \quad \{ \text{ instantiation } y := R \ \} \\
 & (\forall x, y : x \stackrel{+}{\rightarrow} y : b_x \Leftarrow b_y) \\
\Leftarrow & \quad \{ \text{ transitive closure } \} \\
 & (\forall x, y : x \rightarrow y : b_x \Leftarrow b_y) \quad ,
\end{array}
$$

and the formula thus obtained is easily maintained as a system invariant. The assumption $(\forall x : x \neq R : x \stackrel{+}{\rightarrow} R)$ expresses that in the directed subgraph the root is reachable from all other nodes; we will call such a subgraph *rooted*.

Initially $(\forall y :: \neg b_y)$, and to reach the desired final state $(\forall y :: b_y)$, each node y will perform $b_y := \mathsf{true}$. The system invariant $(\forall x, y : x \rightarrow y : b_x \Leftarrow b_y)$ requires that $b_y := \mathsf{true}$ be guarded by $(\forall x : x \rightarrow y : b_x)$, and, as before, this brings about a progress requirement, namely:

$$
(\forall y :: b_y) \quad \vee \quad (\exists y :: \ \neg b_y \ \wedge \ (\forall x : x \rightarrow y : b_x) \ ) \quad ,
$$

which can be rewritten into the equivalent:

$$
(\forall y :: b_y) \quad \Leftarrow \quad (\forall y :: \ b_y \ \Leftarrow \ (\forall x : x \rightarrow y : b_x) \ ) \quad .
$$

Now, this is just the proposition that the relation $\rightarrow$ admits Mathematical Induction! Because the graphs in distributed computations are finite, this is equivalent to the requirement that $\rightarrow$ is *acyclic*, that is, $\stackrel{+}{\rightarrow}$ is irreflexive:

$$
(\forall x :: \neg (x \stackrel{+}{\rightarrow} x) \ ) \quad .
$$

Thus, we conclude that for termination detection a rooted and acyclic directed subgraph of the original graph suffices. Often, a *spanning tree* is used for this purpose, which is a *minimal* such subgraph; the above shows that there is no logical necessity to restrict oneself exclusively to spanning trees.

* * *

Acyclicity of a directed graph is a global property as well, and awkward to maintain as an invariant, for instance, when the directed subgraph is constructed during the distributed computation itself. Therefore, again, we try to reduce it to a more local property first:

$$
\begin{array}{ll}
 & (\forall x :: \neg (x \stackrel{+}{\rightarrow} x) \ ) \\
\equiv & \quad \{ \text{ one-point rule, to introduce } y \text{; range-term trading } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall x,y : x \xrightarrow{+} y : x \neq y \,) \\
\Leftarrow & \quad \{ \text{ introduction of an integer function } f \text{ ; Leibniz } \} \\
 & (\forall x,y : x \xrightarrow{+} y : f_x \neq f_y \,) \\
\Leftarrow & \quad \{ \text{ strengthening } \} \\
 & (\forall x,y : x \xrightarrow{+} y : f_x < f_y \,) \\
\Leftarrow & \quad \{ \text{ transitive closure } \} \\
 & (\forall x,y : x \rightarrow y : f_x < f_y \,) \quad .
\end{array}
$$

The crux of this derivation is the strengthening of the non-transitive relation $\neq$ into the transitive $<$, so as to prepare for the next step; the resulting formula, $(\forall x,y : x \rightarrow y : f_x < f_y \,)$, is much easier to maintain as an invariant.

5 Conclusion

To remain true to the spirit of this symposium, I can think of no better way to conclude this lecture than with a quotation of the master:

> *If it is clumsy, it is not mathematics.*
> Edsger W. Dijkstra

References

[1] R.R. Hoogerwoord, A Formal Development of Distributed Summation, Computing Science Report 00-09, Eindhoven University of Technology, 2000.