1 Calculational Proofs

Vladimir Lifschitz

1.1 A letter from Dijkstra

The letter from Edsger Dijkstra that I hold in my hands, dated 11 December 1996, begins with the words:

> Dear Vladimir,
>
> I did my homework, and proved 1.16 through 1.21 straightforwardly.

The "homework" that Edsger referred to was a set of exercises on the use of the axiomatic method handed out to students in my logic class. Each exercise was a property of natural numbers that could be derived from the assumptions about 0 and the successor function $n \mapsto n'$ expressed by Peano axioms, the recursive definition of addition, and the definition of $m \leq n$ as $\exists k(m + k = n)$. In the letter, Edsger showed how such proofs could be presented in the calculational style, described in his book *Predicate Calculus and Program Semantics*, joint with Carel Scholten [Dijkstra and Scholten 1990]. That book was the outcome of many years of thinking about

> ... the streamlining of the mathematical argument, which was inspired by our observation that many —if not most— mathematical arguments we encountered in the literature were unnecessarily complex and lacking in rigour by being incomplete. A few explorations sufficed to convince us that the potential for improvement was dramatic indeed ... (EWD932a).

Edsger's proof of the formula $m \leq n \vee n \leq m$ shown in Figure 1.1 mentions five other properties of natural numbers:

1.16. $0 \leq n$.
1.17. $n \leq n$.
1.18. $n \leq n'$.
1.20. If $k \leq m$ and $m \leq n$ then $k \leq n$.
1.22. If $m \leq n$ and $m \neq n$ then $m' \leq n$.

It uses induction and consists of two "calculations"—one for the base, the other for the induction step. Each calculation is a list of formulas. Successive members of the list are

1.23, $\langle \forall m,n :: \ m \leq n \vee n \leq m \rangle$, I proved by mathematical induction over m. For the base I observed

$$
\begin{array}{ll}
 & 0 \leq n \vee n \leq 0 \\
\Leftarrow & \quad \{\text{pred. calc.}\} \\
 & 0 \leq n \\
\equiv & \quad \{1.16\} \\
 & \mathit{true}
\end{array}
$$

For the step I observed

$$
\begin{array}{ll}
 & m' \leq n \vee n \leq m' \\
\Leftarrow & \quad \{1.18 \text{ —i.e. } m \leq m' \text{— and } 1.20 \text{ —i.e. transitivity}\} \\
 & m' \leq n \vee n \leq m \\
\Leftarrow & \quad \{1.22 \text{ and } 1.17 + \text{Leibniz: } x = y \Rightarrow y \leq x\} \\
 & (m \leq n \wedge m \neq n) \vee (m = n \vee n \leq m) \\
\equiv & \quad \{\vee \text{ is associative}\} \\
 & ((m \leq n \wedge m \neq n) \vee m = n) \vee n \leq m \\
\Leftarrow & \quad \{\text{predicate calculus}\} \\
 & (m \leq n \vee m = n) \vee n \leq m \\
\Leftarrow & \quad \{1.17 + \text{Leibniz}\} \\
 & m \leq n \vee n \leq m
\end{array}
$$

which completes the proof of 1.23.

Figure 1.1 Dijkstra's calculational proof.

separated by the symbol $\equiv$ ("is equivalent to") or by $\Leftarrow$ ("follows from"), along with a "hint" in braces. [1] The hint explains why the relationship expressed by the symbol indeed holds.

The first calculation shows in two steps how the formula to be proved follows from the formula *true*. The second calculation shows how the consequent $m' \leq n \vee n \leq m'$ of the implication to be proved follows from its antecedent $m \leq n \vee n \leq m$. (Many calculational proofs of implications found in the literature are organized in a different way—as chains of formulas leading from the antecedent to the consequent. In such calculations, the symbol $\Leftarrow$ is replaced by the "implies" symbol $\Rightarrow$.)

Two hints in the second calculation use Edsger's code word for replacing equals by equals, "Leibniz." These two steps use the implications $m = n \Rightarrow n \leq m$ and $m = n \Rightarrow m \leq n$, which are equivalent to 1.17 "by Leibniz."

[1] According to Dijkstra and Misra [2001], this format is due to W.H.J. Feijen.

After a few examples of this kind, Edsger wrote: "I hope I made my point. I think I would like to recommend that you do some experiments with this calculational proof style since it may be a great improvement over what was available."

Although I have not adopted calculational proof style in my own work, I have spent many hours thinking about it, and this chapter is a summary of my conclusions.

1.2 Dijkstra's calculational proofs are semi-formal

Dijkstra and Scholten characterize their calculational proofs as formal. Reasoning in these proofs is similar to formal reasoning, as this term is understood in work on foundations of mathematics, but there is a difference.

Calculational proofs in *Predicate Calculus and Program Semantics* are similar to formal proofs in two ways. First, intermediate results in such a proof are expressed using propositional connectives and quantifiers, and this use of traditional logical notation is crucial; its function is not merely to abbreviate expressions of a natural language. Think, for instance, of nested occurrences of the equivalence symbol, which are found in the book so many times. Natural languages

> are rather ill-equipped for expressing equivalence. We have, of course, the infamous "if and only if" —where "if" takes care of "follows from" and "only if" of implies— but that is no more than an unfortunate patch: by all linguistic standards, the sentence
>
> > "Tom can see with both eyes if and only if Tom can see with only one eye if and only if Tom is blind."
>
> is —probably for its blatant syntactic ambiguity— total gibberish [Dijkstra and Scholten 1990, Chapter 5].

Nested equivalences are hardly ever found in conventional, informal mathematics. Furthermore, calculational proofs, like formal proofs, do not rely on simplifying assumptions ("assume for simplicity", "consider, for instance, the case", "without loss of generality"). They use neither three dots notation $(1, 2, \ldots, 100)$ nor references to pictures.

Second, every step in a calculational proof is a manipulation from some limited repertoire of permissible transitions—as in a formal proof. But in foundational studies, valid justifications for including an assertion in a proof are specified by listing axioms and inference rules, so that the class of formal proofs is defined with mathematical precision, like the class of syntactically correct programs in a programming language. The structure of calculational proofs is described by Dijkstra and Scholten less precisely. Because of this difference, their proofs can be best characterized as semi-formal.

An attempt to identify inference rules that model patterns of calculational reasoning was made by Gries and Schneider [1995], who described a formalization **E** of equational proposi-

tional logic and proved its completeness. Axioms and inference rules of **E** are discussed also by Bijlsma and Nederpelt [1998]. A modification of **E** proposed a few years later [Lifschitz 2002] is reviewed in Section 1.5 below; that article shows how adding inference rules for quantifiers can turn **E** into a complete deductive system of first-order logic.

1.3 Calculational proofs can be smooth

When Dijkstra and Scholten started experimenting with the calculational style, each proof was for them "a well-crafted, but isolated, piece of ingenuity"; later on they developed

> heuristics from which most of the arguments emerged most smoothly (almost according to the principle "There is really only one thing one can do."). The heuristics turned the design of beautiful, formal proofs into an eminently teachable subject [Dijkstra and Scholten 1990, Chapter 0].

The idea of smooth argument—reasoning without surprising steps— played an important role in Edsger's thinking. In a note on mathematical methodology (EWD1067), after presenting a solution to a puzzle, Edsger writes: "Some may think this solution cute, surprising, or ingenious, but I would like to point out that there is nothing ingenious about it, because the argument is all but forced." For Edsger in the late eighties, ingenious was bad. In the chapter on the proof format, the authors give examples of calculational proofs containing the implication symbol ($\Rightarrow$), then introduce the consequence symbol ($\Leftarrow$), and explain:

> Before we introduced the consequence, we had many calculations that required considerable clairvoyance to write down in the sense that the motivation for certain manipulations would become apparent several lines further down, where everything would miraculously fall into place. Upon reading them they struck us each time as if a few rabbits had been pulled out of a hat. As soon as we realized that we ourselves had designed those proofs the other way round, we decided to present them in that direction as well, and the symbol $\Leftarrow$ was introduced. Suddenly, many a manipulation was now strongly suggested by what had already been written down. We were almost shocked to see how great a difference such a trivial change in presentation could make and came to fear that the traditional predominance of the implication over the consequence in combination of our habit of reading from left to right has greatly contributed to the general mystification of mathematics [Dijkstra and Scholten 1990, Chapter 4].

Proofs of identities that we remember from high school algebra are examples of good proofs from this perspective. We keep simplifying one side of the identity until we arrive at the other side:

$$(a+b)(a-b) = a^2 + ab - ab - b^2 = a^2 - b^2.$$

Symbols do the job; every step is forced; there is nothing surprising or ingenious. Good calculational proofs are similar to this calculation, except that logical formulas take the place of polynomials.

Many mathematicians would be puzzled by the view that smooth reasoning is good mathematics. We prove that there are infinitely many primes by observing that for any list $p_1, \ldots, p_n$ of primes, prime divisors of

$$p_1 \cdot \ldots \cdot p_n + 1 \tag{1.1}$$

do not belong to the list. Expression (1.1) is a rabbit out of a hat, and this rabbit is what makes the proof breathtakingly beautiful. This kind of beauty can be found in Edsger's own work. He surprised us, for instance, by proving a theorem from number theory using a graph with words attached to its vertices (EWD740). That proof is discussed in Chapter **??**, written by Jayadev Misra.

My guess is that Edsger's experience with program development has led him to the conclusion that surprising steps are unnecessary and harmful when mathematics is used to develop a program along with the proof of its correctness. That would explain why he considered smooth reasoning so important.

1.4 Calculational proofs vs. natural deduction

Natural deduction codifies the use of assumptions in mathematical reasoning. Its invention was motivated by desire

> to construct a formalism that comes as close as possible to actual reasoning. Thus arose a "calculus of natural deduction" [Gentzen 1934].

Derivable objects in a natural deduction system are pairs—a formula F along with a set Γ of assumptions under which F is asserted to hold. Inference rules in such a deductive system are classified into introduction and elimination rules:

> $\wedge$-*introduction:* If F holds under assumptions Γ and G holds under assumptions Δ then conclude that $F \wedge G$ holds under assumptions $\Gamma \cup \Delta$.
>
> $\wedge$-*elimination:* If $F \wedge G$ holds under assumptions Γ then conclude that F and G hold under assumptions Γ.
>
> $\Rightarrow$-*introduction:* If G holds under assumptions Γ then conclude that $F \Rightarrow G$ holds under assumptions $\Gamma \setminus \{F\}$.
>
> $\Rightarrow$-*elimination:* If $F \Rightarrow G$ holds under assumptions Γ and F holds under assumptions Δ then conclude that G holds under assumptions $\Gamma \cup \Delta$.

Constructing a natural deduction proof can be planned according to heuristic rules:

H1. If you want to derive a conjunction $F \wedge G$ then derive F and G, and then use $\wedge$-introduction.

H2. If you assumed or derived $F \wedge G$ then use $\wedge$-elimination to derive F and G.

H3. If you want to derive an implication $F \Rightarrow G$ then assume F, derive G, and then use $\Rightarrow$-introduction.

H4. If you assumed or derived $F \Rightarrow G$ then derive also F, and then use $\Rightarrow$-elimination to derive G.

For example, if we assumed or derived a formula of the form $F \wedge G \Rightarrow H$ then H4 recommends that we try to derive $F \wedge G$ and then use $\Rightarrow$-elimination to derive H. The goal of deriving $F \wedge G$ may be achieved, according to H1, by deriving F and G, and then using $\wedge$-introduction.

To give another example, if our goal is to derive a formula of the form $F \Rightarrow (G \Rightarrow H)$ then H3 recommends that we assume F and try to use this assumption to derive $G \Rightarrow H$. The goal of deriving $G \Rightarrow H$ may be achieved, according to H3, by assuming G and then using this second assumption, along with the assumption F, to derive H. This will be followed by two $\Rightarrow$-introduction steps.

Advice on designing proofs that Dijkstra and Scholten give in their book looks very different—they tell us how to better organize a calculation. For example, they talk about choosing between two possible ways to plan a calculational proof of an implication: is it better to go from the antecedent to the consequent using $\equiv$ and $\Rightarrow$ transitions or to go from the consequent to the antecedent using $\equiv$ and $\Leftarrow$ transitions?

> Experience has taught us that in general the most complicated side is the most profitable one to start with. The probable explanation of this phenomenon is that the opportunities for simplification are usually much more restricted than the possibilities to complicate things: simplification is much more opportunity-driven than "complification" [Dijkstra and Scholten 1990, Chapter 4].

In the letter quoted above, Edsger gave another advice: he observed that

$$m = n \Rightarrow m \leq n$$

is "a more useful statement of the reflexivity of "$\leq$" than $n \leq n$: formulae with more (universally quantified) dummies admit more instantiations and are, hence, a more flexible tool."

In one important case the calculational approach suggests a strategy different from the natural deduction strategy outlined above. An equivalence $F \equiv G$ can be viewed as shorthand

for the conjunction

$$(F \Rightarrow G) \wedge (G \Rightarrow F),$$

so that a proof of this equivalence, according to heuristic rules H1 and H3, may be a "ping-pong" argument: assume F and derive G, and then assume G and derive F. The preferred calculational method of proving an equivalence is to construct a chain of equivalent formulas connecting the left-hand side with the right-hand side; there is no need then to prove each of the two implications separately.

Dijkstra and Scholten note that

> [m]any texts on theorem proving seem to suggest that a ping-pong argument is —if not the only way— the preferred way of demonstrating equivalence. This opinion is not confirmed by our experience: avoiding unnecessary ping-pong arguments has turned out to be a powerful way of shortening proofs. At this stage methodological advice is to avoid the ping-pong argument if you can, but to learn to recognize the situations in which it is appropriate [Dijkstra and Scholten 1990, Chapter 4].

I found this advice most helpful: my own proofs of if-and-only-if statements are often chains of equivalences, although they do not strictly follow the Feijen—Dijkstra—Scholten format. [2]

It would be a mistake, however, to think that calculational proofs have nothing in common with natural deduction. Heuristic rules H2 and H4 tell us how a useful next step in a natural deduction proof is determined by the shape of the formula produced in the previous step, which is a feature of smooth calculational proofs that Edsger valued so much. Furthermore, consider his calculational proof shown in Figure 1.1. It consists of a reference to "mathematical induction over m" followed by two calculations. The induction axiom, which is not stated there explicitly, has the form

$$\mathit{Base} \wedge \mathit{Step} \Rightarrow \mathit{Conclusion}.$$

The first calculation proves *Base*; the second proves *Step*. The assertion that providing these calculations completes the proof of 1.23 looks pretty much like the use of $\wedge$-introduction to conclude $\mathit{Base} \wedge \mathit{Step}$, followed by the use of $\Rightarrow$-elimination to arrive at *Conclusion*.

Note also that the proof strategy used in establishing *Base* in this example—building a chain of formulas leading from the goal to the formula *true*—is applicable, in principle, to goal formulas of any syntactic form, including implications. But calculational proofs of implications are usually organized in a different way: they are chains leading from the antecedent to the consequent (or the other way around). This looks pretty much like the use of heuristic rule H3.

[2] See, for instance, the proof of Lemma 8 in the article by Ferraris et al. [2011].

1.5 Propositional calculations

We will show now that some calculations involving propositional formulas correspond to derivations in equational propositional logic.

In this section, *formulas* are propositional formulas formed from atoms and the 0-place connective $\bot$ ("false") using the binary connectives $\equiv$ and $\vee$. Other connectives are treated as abbreviations; for instance, $\neg F$ is shorthand for $F \equiv \bot$.

The axiom schemas of the deductive system **E** express that equivalence is commutative

$$(F \equiv G) \equiv (G \equiv F)$$

and associative

$$(F \equiv (G \equiv H)) \equiv ((F \equiv G) \equiv H);$$

disjunction is commutative

$$F \vee G \equiv G \vee F,$$

associative

$$F \vee (G \vee H) \equiv (F \vee G) \vee H$$

and idempotent

$$F \vee F \equiv F;$$

it distributes over equivalence

$$F \vee (G \equiv H) \equiv (F \vee G \equiv F \vee H),$$

and $\bot$ is its neutral element

$$F \vee \bot \equiv F.$$

The system **E** has two inference rules. One of them, when applied to premises F and $G \equiv H$, gives a formula obtained from F by replacing some (or all) occurrences of G by H. The effect of the second rule is opposite—it replaces occurrences of H by G. These rules can be written as

$$\frac{F(G) \quad G \equiv H}{F(H)} \qquad \text{and} \qquad \frac{F(H) \quad G \equiv H}{F(G)} \ .$$

(By $F(G)$ we denote the result of substituting G for all occurrences of the placeholder atom X in $F(X)$.) The simpler rules

$$\frac{G \quad G \equiv H}{H} \qquad \text{and} \qquad \frac{H \quad G \equiv H}{G} \ ,$$

are special cases (when $F(X)$ is X).

$$\dfrac{\dfrac{\dfrac{\neg F \equiv G \qquad (F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)}{F \equiv (\bot \equiv G)} \qquad (\bot \equiv G) \equiv \neg G}{F \equiv \neg G} \qquad (F \equiv \neg G) \equiv \neg (F \equiv G)}{\neg (F \equiv G)}$$

Figure 1.2 A derivation of $\neg(F \equiv G)$ from the assumption $\neg F \equiv G$. Formulas at three leaves are axioms: $(F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)$ and $(F \equiv \neg G) \equiv \neg(F \equiv G)$ are special cases of the associativity of equivalence; $(\bot \equiv G) \equiv \neg G$ is a special case of the commutativity of equivalence.

We will represent derivations in the system **E** as trees, with axioms and assumptions at leaves. For example, Figure 1.2 is a derivation of $\neg(F \equiv G)$ from the assumption $\neg F \equiv G$.

A derivation in the system **E** is *simple* if each of its formulas that does not belong to the leftmost branch is a leaf. Thus a simple derivation is a single branch with several leaves attached to it. For example, the derivation in Figure 1.2 is simple. Simple derivations have the form

$$\dfrac{\dfrac{\dfrac{\dfrac{F_1 \qquad G_1 \equiv H_1}{F_2} \qquad G_2 \equiv H_2}{\cdots}}{F_{n-1}} \qquad G_{n-1} \equiv H_{n-1}}{F_n}\,. \tag{1.2}$$

The formula F_{i+1} is obtained from F_i either by replacing occurrences of G_i by H_i or by replacing occurrences of H_i by G_i ($i = 1, \ldots, n-1$).

It is easy to note that simple derivations are disguised calculations in the sense of Dijkstra and Scholten. Derivation (1.2) can be viewed as a chain of transformations

$$
\begin{array}{ll}
 & F_1 \\
\equiv & \quad \{G_1 \equiv H_1\} \\
 & F_2 \\
\equiv & \quad \{G_2 \equiv H_2\} \\
\cdots & \\
 & F_{n-1} \\
\equiv & \quad \{G_{n-1} \equiv H_{n-1}\} \\
 & F_n
\end{array}
$$

that lead from F_1 to F_n and are justified by the hints $G_i \equiv H_i$. For example, the derivation in Figure 1.2 can be viewed as a calculation establishing the equivalence between $\neg F \equiv G$ and $\neg(F \equiv G)$ using axioms of **E** as hints:

$$
\begin{array}{ll}
 & \neg F \equiv G \\
\equiv & \quad \{(F \equiv (\bot \equiv G)) \equiv (\neg F \equiv G)\} \\
 & F \equiv (\bot \equiv G) \\
\equiv & \quad \{(\bot \equiv G) \equiv \neg G\} \\
 & F \equiv \neg G \\
\equiv & \quad \{(F \equiv \neg G) \equiv \neg(F \equiv G)\} \\
 & \neg(F \equiv G).
\end{array}
$$

1.6 Reasoning about predicate transformers

Dijkstra and Scholten begin their book with a discussion of notational conventions that we have not yet talked about—conventions that

> ... may strike the reader as a gross overloading of all sorts of familiar operators: for instance, we apply the operators from familiar two-valued logic to operands that in some admissible models may take on uncountably many distinct values [Dijkstra and Scholten 1990, Chapter 1].

This aspect of their contribution is essential for the study of predicate transformers, such as the weakest precondition operator.

A *(unary) predicate* on a domain D is a function that maps elements of D to truth values *false*, *true*. We will identify a predicate P with the set of elements of the domain that are mapped by P to *true*. Thus predicates can be treated as subsets of the domain, and we can write $u \in P$ unstead of $P(u)$; this is sometimes convenient. A *predicate transformer* is a function

that maps predicates to predicates. [3] For example, if the domain is the set of all integers then a predicate transformer f can be defined by the formula

$$f(P) = \{u : u+1 \in P\}. \tag{1.3}$$

The domain of integers can be viewed as the state space of programs that involve a single integer variable. We can say that the predicate transformer f defined by (1.3) is the weakest precondition operator for the assignment $u := u+1$.

The following example illustrates the use of overloading. A predicate transformer f is *monotonic* if for all predicates P, Q

$$\forall u(P(u) \Rightarrow Q(u)) \Rightarrow \forall u((f(P))(u) \Rightarrow (f(Q))(u)), \tag{1.4}$$

where u is a variable that ranges over elements of the domain. The Dijkstra-Scholten overloading convention allows us to apply the connective $\Rightarrow$ to predicates, rather than truth-valued expressions, and we can write $P \Rightarrow Q$ to represent the predicate $\{u : P(u) \Rightarrow Q(u)\}$. Furthermore, the square brackets around a predicate expression (the "everywhere operator") convert it into the assertion that the predicate maps all elements of the domain to *true*. The antecedent of (1.4) can be written using this notation as $[P \Rightarrow Q]$. Similarly, the consequent can be written as $[f(P) \Rightarrow f(Q)]$. Thus formula (1.4) turns into the much shorter expression

$$[P \Rightarrow Q] \Rightarrow [f(P) \Rightarrow f(Q)].$$

This is how Dijkstra and Scholten express monotonicity of predicate transformers on page 28 of their book (except that they denote functional application by an infix full stop, so that $f(P)$ becomes $f.P$).

As noted by Rutger Dijkstra [1996] and by Gries and Schneider [1998], the everywhere operator is similar to modal operators, which can be understood as quantifiers binding variables for possible worlds [Garson 2021, Section 6].

To take another example of the use of overloading, a predicate transformer f is *conjunctive* if for all predicates P, Q

$$\forall u((f(R))(u) \equiv (f(P))(u) \wedge (f(Q))(u)), \tag{1.5}$$

where R stands for $\{u : P(u) \wedge Q(u)\}$. Overloading $\equiv$ and $\wedge$ allows us to write (1.5) as

$$[f(R) \equiv f(P) \wedge f(Q)].$$

The expression that R stands for can be written as $P \wedge Q$, so that (1.5) turns into

$$[f(P \wedge Q) \equiv f(P) \wedge f(Q)].$$

[3] In Chapter **??**, written by Reiner Hähnle, given a program, predicate transformers are understood as mappings from formulas to formulas, so that they operate with syntactic objects representing subsets of the domain.

We observe that for any conjunctive predicate transformer f and any predicates P , Q

$$
\begin{array}{ll}
 & [f.P \Rightarrow f.Q] \\
\equiv & \quad \{\text{predicate calculus}\} \\
 & [f.P \wedge f.Q \equiv f.Q] \\
\equiv & \quad \{\ f \text{ is conjunctive}\} \\
 & [f.(P \wedge Q) \equiv f.Q] \\
\Leftarrow & \quad \{\text{Leibniz}\} \\
 & [P \wedge Q \equiv Q] \\
\equiv & \quad \{\text{predicate calculus}\} \\
 & [P \Rightarrow Q]
\end{array}
$$

Figure 1.3 Proof of the theorem: "A conjunctive predicate transformer is monotonic" [Dijkstra and Scholten 1990, Chapter 4].

A calculational proof that uses overloading and the everywhere operator is shown in Figure 1.3. There are no quantifiers in this calculation, but its validity can be verified by applying equivalent transformations to formulas with quantifiers. For instance, the last of the four hints, "predicate calculus," indicates the use of the formula

$$[P \Rightarrow Q] \equiv [P \wedge Q \equiv P],$$

which can be understood as shorthand for the equivalence

$$\forall u(P(u) \Rightarrow Q(u)) \equiv \forall u(P(u) \wedge Q(u) \equiv P(u)).$$

This equivalence is indeed a theorem of predicate calculus. The authors do not teach us, however, to work with variables ranging over elements of the domain. Instead, they refer the reader to the chapter entitled "The calculus of boolean structures," which includes a long list of useful equivalences that contain the everywhere operator but no explicit quantifiers.

One of the symbols to which Dijkstra and Scholten apply their overloading convention is equality, and this particular deviation from tradition may be confusing. Propositional connectives are traditionally applied to truth-valued expressions, and the reader naturally looks for an explanation whenever they are applied to expressions of any other kind. With the equality symbol, the situation is different: the equality $P = Q$, where P and Q are predicates, has a standard meaning, which is the same as the meaning of $[P = Q]$ in the book. This mismatch may be one of the reasons why many authors found Edsger's use of brackets hard to understand, as discussed in Chapter **??**, written by David Gries.

1.7 Conclusion

The invention of calculational proofs did not revolutionize mathematical writing, as Edsger had perhaps hoped. In fact, a detailed review of *Predicate Calculus and Program Semantics* in *Science of Computer Programming* was highly critical ("the book will not be helpful to those interested in the subject area") [Börger 1994]. But reading the book has helped many computer scientists understand what a well-written proof is and to learn the art writing good proofs.

In the letter quoted at the beginning of this chapter, Edsger wrote that he saw calculational style

> more and more being adopted: first that was in Ph.D. theses (of candidates I did not know, supervised by colleagues I hardly knew), lately I saw it completely adopted in the textbook "Algebra of Programming" by Bird and de Moor. That book shows another symptom of the acceptance: it just displays its proofs in this format without explaining it and without giving credit for it to the people who designed and promoted it.

Calculational proofs have indeed become standard in work on the formal development of programs [Back and von Wright 1998, Backhouse 2003, Kaldewaij 1990]. A textbook that is widely used in undergraduate teaching [Gries and Schneider 1993] has examples of Dijkstra-style calculations on nearly every page. In the preface the authors testify that, in their experience, "an equational logic, which is based on equality and Leibniz's rule for substitution of equals for equals" is the logic best suited to be used as a tool in every-day work. I can add that it was a pleasure for me to write proofs in collaboration with students at the University of Texas who were fortunate to have taken Edsger's classes.

Acknowledgements

Many thanks to David Gries, Jayadev Misra and Fred Schneider for valuable comments on a draft of this chapter.

Bibliography

R. Back and J. von Wright. 1998. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer.

R. Backhouse. 2003. *Program Construction - Calculating implementations from specifications*. John Wiley.

L. Bijlsma and R. Nederpelt. 1998. Dijkstra–Scholten predicate calculus: concepts and misconceptions. *Acta Informatica*, 35: 1007–1036.

E. Börger. 1994. Book review: E.W.Dijkstra, C.S.Scholten, Predicate calculus and program semantics. *Science of Computer Programming*, 23: 1–11.

E. Dijkstra and C. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag.

E. W. Dijkstra and J. Misra. 2001. Designing a calculational proof of Cantor's theorem. *The American Mathematical Monthly*, 108(5): 440–443.

R. Dijkstra. 1996. Everywhere in predicate algebra and modal logic. *Information Processing Letters*, 58: 237–243.

P. Ferraris, J. Lee, and V. Lifschitz. 2011. Stable models and circumscription. *Artificial Intelligence*, 175: 236–263.

J. Garson. 2021. Modal logic. In E. N. Zalta, ed., *The Stanford Encyclopedia of Philosophy (Summer 2021 Edition)*.

G. K. E. Gentzen. 1934. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39: 176–210.

D. Gries and F. Schneider. 1993. *A Logical Approach to Discrete Math*. Springer.

D. Gries and F. Schneider. 1995. Equational propositional logic. *Information Processing Letters*, 53: 145–152.

D. Gries and F. Schneider. 1998. Adding the everywhere operator to propositional logic. *Journal of Logic and Computation*, 8: 119–129.

A. Kaldewaij. 1990. *Programming - the derivation of algorithms*. Prentice Hall international series in computer science. Prentice Hall.

V. Lifschitz. 2002. On calculational proofs. *Annals of Pure and Applied Logic*, 113: 207–224.