What Programmers Don't and Should Know†

David Gries

87-872

September 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†Prepared for the Twentieth University of Newcastle upon Tyne International Seminar on the Teaching of Computing Science at University Level, titled *Logic and its Application to Computing Science.*

What Programmers Don't and Should Know

David Gries
Computer Science Department, Cornell University

prepared for the

Twentieth University of Newcastle upon Tyne International Seminar
on the Teaching of Computing Science at University Level

titled

Logic and its Application to Computing Science

8-11 September 1987

Table of Contents

Introduction

I am concerned with the way algorithms and programs are developed and presented in the research papers that we write, in the texts from which we teach, and in 'everyday' programming. Generally speaking, the level of professionalism is far below what it could be, given the advances in the state of the art and science of programming over the past fifteen years,

It is doubtful that older computer scientists will change their methods of developing and presenting algorithms. The development and presentation of an algorithm is often the secondary concern to them; the primary concern is the area in which the algorithm is being written, and that is where most of the learning and research efforts are placed. (Also, it is difficult, for all of us, to change our ways —I certainly don't want to. As Mark Twain sai
'Nothing so needs reforming as other people's habits'.)

Thus, it is to the younger generations that we turn fc improvement. If we teach them more effective ways c thinking, of developing and presenting algorithms, they i turn will press for and effect the necessary changes.

But what should we give today's students that they are n already getting? First, we should be instilling in them

• An appreciation of the need for precision, rigor, an elegance when dealing with specifications of an algc rithm, when analyzing the properties of the objects bein manipulated by the algorithm, when developing it, an when presenting it.

Of course, an appreciation of a need is not enough; or must be able to *fill* the need. Basic to this is

• An in-depth experience with the propositional an predicate calculi, the aim being a proficiency and agilit in formal manipulation according to their axioms, infe ence rules, and theorems.

The emphasis here is not on deep theorems about co sistency, completeness, non-standard models, and the lik Instead, it is on the use of logic as a tool in our everyda striving for simplicity and elegance. The *syntactic pro* as a sequence of formal manipulations according to logic should assume importance. Through many an varied examples, it should become clear that formal manipulation is a useful —even indispensable— *tool* of the programmer. Out of this will come

• An appreciation of the importance of suitable, simple notation that is geared to formal manipulation and a lessening of the need for examples as a means of conveying understanding.

One may argue that these things are already being taught and practiced. Perhaps, but far less than they should be. In the U.S., they are to be conveyed in a discrete mathematics course, where the student is first introduced to the predicate calculus, but in texts for these courses the emphasis is more on facts and knowledge than on method and appreciation. Rarely is the predicate calculus used outside the single chapter devoted to it, rarely is there a discussion and comparison of notations, and rarely are different proofs or proof methods compared.

Further, relatively few articles and texts on data structures and algorithms make use of what we know about programming and the description of algorithms, so any appreciation absorbed earlier is not reinforced.

In summary, more emphasis should be placed on mathematical tools and methods, and in such a way that an appreciation for their need and for elegance and simplicity, as well as a sense of discrimination, is inculcated in the student. Below, I want to explain in more detail what I mean, using examples from both mathematics and programming and giving reasons why one method or notation might be preferred over others. Formal methods for program correctness are of course discussed, but the student needs to master some basic mathematics before these can be appreciated and applied.

We begin with a brief overview of the propositional and predicate calculi.

1. Logic

Two reasons for using a formal system of logic in our work are:

• To be able to state concepts (for example, mathematical induction) or statements (for example, a specification of a program) clearly and unambiguously.

• To be able to give shorter, simpler, and more elegant proofs and derivations and to increase our powers of reasoning; the proofs are for human, rather than machine, consumption.

Let us discuss logic briefly in light of these reasons.

The propositional calculus

Basic to the propositional calculus is a set of axioms an inference rules that allow the manipulation of formula A *theorem* is an axiom or a formula that can be generate from the axioms using the inference rules. I see two ge eral approaches to such proofs, equational reasoning an natural deduction.

In the equational-reasoning approach, the main inferenc rules include substitution of equals for equals, transitivi of equality, and modus ponens. These rules allow us t use axioms and theorems like De Morgan's La $(\neg(X \vee Y) \equiv \neg X \wedge \neg Y)$ and Associativity of Disjun tion to translate one formula into an equivalent one (into one that is implied by the first). For example, the fo lowing proof shows, by the law of transitivity of equalit that $b \Rightarrow c \equiv \neg c \Rightarrow \neg b$:

$$\begin{array}{lll} & b \Rightarrow c & \\ \equiv & \neg b \vee c & \text{(Implication)} \\ \equiv & c \vee \neg b & \text{(Commutativity)} \\ \equiv & \neg\neg c \vee \neg b & \text{(Double negation)} \\ \equiv & \neg c \Rightarrow \neg b & \text{(Implication)} \end{array}$$

Each line of the proof follows from the previous one by substitution of equals for equals using the axiom (theorem of equality given to the right . One often us more than one rule in going from one line to the nex attempting to achieve brevity without sacrificing unde standing.

Remark. Logicians tend to use the symbol $\supset$ for implic tion. Using X and Y to denote the sets of states in which and y are true, we have $x \supset y$ iff $X \subset Y$, which is indeed confusing use of symbols. Thus, we prefer a differe symbol for implication. □

The proof can also be written as follows, where each ne formula may be preceded by a line that describes the re son the transformation is valid; this is useful when the re son may be more complicated and may take more space.

$$\begin{array}{ll} & b \Rightarrow c \\ \equiv & \{\text{Use the Law of Implication}\} \\ & \neg b \vee c \\ \equiv & c \vee \neg b \\ \equiv & \{\text{Double negation}\} \\ & \neg\neg c \vee \neg b \\ \equiv & \neg c \Rightarrow \neg b \end{array}$$

In a *natural deduction* system, introduced by Gerha Gentzen in the 1930's, no formulas are assumed to l axiomatically valid; there are only rules of inference. 7 compensate for the lack of axioms, it is permitted to intr duce any formula as a hypothesis at any stage. Generall such a system has two kinds of inference rules for ea operator, introduction rules and elimination rules. O kind introduces the operator, the other eliminates it. F example, the rules for $\wedge$ are

$$\wedge\text{-I: } \frac{X, Y}{X \wedge Y} \quad \text{and} \quad \wedge\text{-E: } \frac{X \wedge Y}{X}, \frac{X \wedge Y}{Y}$$

The first rule indicates that, for any formulas X and Y, from the assumptions X and Y the formula $X \wedge Y$ can be inferred. The other two indicate that from $X \wedge Y$ both X and Y can be inferred.

As in the equational-theory approach, a proof consists of a sequence of lines; each is an assumption, an axiom, or an instance of the result of an inference rule for which the corresponding instances of the hypotheses appear on previous lines of the proof. To the right is stated the inference rule being used and the numbers of previous lines containing its assumptions. Here is a proof:

From $p \wedge q$ infer $q \wedge p$

1	$p \wedge q$	assumption 1
2	p	$\wedge$-E, 1
3	q	$\wedge$-E, 1
4	$q \wedge p$	$\wedge$-I, 3, 2

The natural deduction system is thus called because it is supposed to mimic the way we 'naturally' reason (perhaps that is a good reason for eschewing it!), implying that we naturally think of introducing and eliminating operators. Nevertheless, few mathematicians and computing scientists use such inference rules as a formal tool in their work, for they are just too cumbersome. However, the approach has been extremely useful for *studying* logic and is becoming more and more useful in mechanical theorem proving and, for this reason, the computing scientist should be familiar with it. And some systems for doing formal mathematics on the computer are indeed becoming very useful tools and could be helpful in teaching students about formalism and its uses; I refer for example to Constable's system *PRL* [9].

One can introduce the substitution-of-equals-for-equals rule as a meta-rule of a natural-deduction system, thus merging the equational and natural-deduction approaches. Therefore, we need not worry about which is more powerful, etc. Instead, we should be looking at how informal proofs can be written using a mixture of the two methods to arrive at the best proofs. In general, the equational approach does tend to lead to shorter, more readable, proofs.

The above proofs are *syntactic* in nature, because they are simply a syntactic manipulation of formulas without regard to their meaning. My opinion is that we should be striving more and more for such syntactic proofs, and I will have more to say on this later.

A few colleagues have mentioned that *formal* might be more appropriate than *syntactic*, in the sense I am using it. I prefer *syntactic* because it emphasizes more the complete abstraction of the meaning of the symbols th are being manipulated in a proof, an important propert that has to be made clear.

Our use of the propositional calculus fits in nicely wit our notion of computers and states. During execution of program, the computer is in a *state*, which contains value for each variable. The state is therefore a functio from variables to values. Using s to denote the state, i variable x has value v in state s, then $s.x = v$. Thus, th notion of a *model* for the calculus arises naturally. Unfo tunately, programmers are usually taught *only* the mode in that they are taught how to evaluate Boolean expres sions but not the rules for manipulating them.

Remark. I use '.' to denote function application, as a experiment to see whether its use reduces the number o parentheses in formulas, thus making manipulation easie Function application binds tightest, so $f.x+2 = (f.x)+2$ We write the application of a function of two argument as $g.(x,y)$ or, using currying, as $g.x.y$. □

The predicate calculus

The introduction of predicates allows variables of othe types to be used, and with this we introduce quantifie expressions. My discussion here deals mainly with th notation used for quantified expressions and the calculu of axioms and inference rules to be used.

Typically, mathematicians use a notation like $\exists x.\ P$, o perhaps $\exists x \in X.\ P$, to stand for 'there exists a value such that P holds'. Following (but deviating slightl from) Dijkstra, I prefer instead the notation

(0) $\quad \exists(x\colon R\colon P)$

where R is a predicate specifying the range of values under consideration. Actually, (0) can be written in th more conventional notation as $\exists x.\ R \wedge P$, so I need t substantiate my use of it. There are several reasons for it.

First and foremost, our notation should be geared to ou manipulative needs. Often, in programming and relate fields, we manipulate quantified expressions in which th range R remains constant —e.g. it is the subscript rang of an array— while P changes. Making the range distinc allows us to show this more clearly. It allows us to intr duce conventions to eliminate the range, thus reducin what has to be written. For example, an omitted range i assumed to be the same as on the previous line, as in

$$\begin{aligned} & \exists(x\colon R\colon P) \\ \equiv\ & \exists(x\colon\colon P') \end{aligned}$$

Also, we often find ourselves *splitting* a range:

$$\begin{aligned} & \forall(i\colon 0 \le i < n+1\colon P) \equiv \\ & \forall(i\colon 0 \le i < n\colon P) \wedge P^x_n, \end{aligned}$$

and using form (0) instead of the form $\exists x.R \wedge P$ allows us to develop inference rules for manipulating the range. Thus, we are choosing notations geared to our manipulative needs and suggestive of our problematic concerns.

Remark on notation for textual substitution. I am using R_e^x to denote a copy of R with all free occurrences of x replaced by e. Many other notations are used for textual substitution, including R_x^e, $R(x/e)$ and $R(e/x)$. For a linear notation $R[x:=e]$ is a suitable choice because of the connection between assignment and textual substitution; the assignment statement axiom would read

$$\{R[x:=e]\}\ x:=e\ \{R\}$$

Next, we require the parentheses in the new notation because of the importance of the scope-introducing concept. The beginning of the scope of the newly introduced variable is usually clear; the end of the scope should be just as clear.

Finally, let me discuss the use of the quantifier outside the parentheses. We can view the notation

$$x:\ R:\ P$$

as simply a *scope-introducing* mechanism, in which x is the variable introduced, R is a predicate giving its range, and P is some expression or statement, possible containing free occurrences of x. We can apply operators to the scope:

0. $\exists(x:\ R:\ P)$
1. $\forall(x:\ R:\ P)$
2. $\mathbf{N}(x:\ R:\ P)$
3. $\Sigma(x:\ R:\ E)$
4. $\Pi(x:\ R:\ E)$
5. $MAX(x:\ R:\ E)$
6. $\lambda(x:\ R:\ E)$
7. $BLOCK(x:\ R:\ S)$
8. $\textbf{for}(x:\ R:\ S)$
9. $\{x:\ R:\ E\}$

In cases 0-2, P is of type Boolean. Case 2 denotes the number of values x in the range R such that P holds. In cases 3-5, E is integer or real-valued; these produce the sum, the product, and the maximum over values E such that R holds. Case 6 is a function that yields the value E_a^x given an argument a. Case 7 is an Algol-like block that introduces a new variable; case 8 is a loop, executing statement S for all values of x in the range R. Case 9 is the set of values E where x ranges over values for which R is true.

It is the separation of the range R from the expression P and the explicit introduction of the fresh variable, the dummy, that allows us to unify in this fashion.

Operator $\mathbf{N}$ (case 2) can be used to simplify what might otherwise be very awkward. For example, the following statement says that sequence b is a permutation of sequence c:

$$\forall(v::\ \mathbf{N}(i:\ 0\le i<\#b:\ b.i=v) = \mathbf{N}(i:\ 0\le i<\#c:\ c.i=v))$$

i.e. each value occurs the same number of times in b an c. Try to express this statement without using operator N

Suitable axioms for quantification

The student must learn the rules for manipulating quanti fied expressions. These seem difficult at first, but practic quickly leads to internalization. Useful axioms an theorems of equivalence, geared to our manipulativ needs, appear in [2] and [3]. We list some for the quanti fier $\forall$. We will be using such axioms subsequently.

0. **De Morgan**: $\neg\forall(x:\ P:\ Q) \equiv \exists(x:\ P:\ \neg Q)$
1. $\forall(x:\ false:\ P)$
2. $\forall(x:\ P \wedge Q:\ R) \equiv \forall(x:\ P:\ \neg Q \vee R)$
3. $\forall(x:\ P:\ R) \Rightarrow \forall(x:\ P:\ Q \vee R)$
4. **Dummy renaming**:
   $\forall(x:\ P:\ Q) \equiv \forall(y:\ P_y^x:\ R_y^x)$
   (where y is a fresh variable)
5. **One-point rule**: $\forall(x:\ x=E:\ P) \equiv P_E^x$
   (where x is not free in E).
6. **Range-splitting**: $\forall(x:\ P \vee Q:\ R) \equiv$
   $\forall(x:P:R) \wedge \forall(x:Q:R)$

2. Writing formulas in the logical notation

The student must begin to feel that logic is useful: h needs *motivation*. The presentation of many examples c the formalization of statements and concepts will hel We give some examples here.

Induction and well-foundedness

First, let us consider the concept of complete mathemat cal induction. Typically, this is stated as follows: Let P. be a predicate with argument x, where x ranges over th natural numbers. Suppose we prove the base case: P. holds. Suppose we prove the inductive case: for all $y >$ (if $P.x$ holds for all x less than y then $P.y$ holds. Then w conclude that $P.x$ holds for all natural numbers.

We now generalize and formalize this statement. First, l U be a set of values and let $<$ be any binary relation ove U. We say that $(U, <)$ *admits induction* if the followin holds. Let $P.x$ be a predicate, with x ranging over U Then $P.x$ holds for all $x \in U$ iff, for any y, the truth of P. for all x less than y implies the truth of $P.y$. We formal ize this as follows:

(1) **Complete mathematical induction**:
$\forall(x\colon x \in U\colon P.x) \equiv$
$\forall(y\colon y \in U\colon \forall(x\colon x<y\colon P.x) \Rightarrow P.y)$

The equivalent formula (2) does not use implication. To show that (1) $\equiv$ (2), use the Laws of Implication, De Morgan, and Commutativity of $\vee$. In (2) and subsequent discussions of induction, the range of a quantified expression is omitted if it is the universe U.

(2) **Complete mathematical induction**:
$\forall(x:: P.x) \equiv$
$\forall(y:: P.y \vee \exists(x\colon x<y\colon \neg P.x))$

Note that these formulas are equivalences and not implications, although the original informal statement of induction over the natural numbers was couched as an implication. The stronger and commutative equivalence is preferred over the weaker and non-commutative implication.

Note also that (1) and (2) do not distinguish between the base case and the inductive case. In general, avoid case analysis like the plague; even reducing two cases to one is a worthwhile simplification.

It has been claimed that induction should be described in terms of a base case and an induction case because that's how we use it. Before agreeing with this claim, decide whether our traditional two-case view has been forced on us by the two-case formulation. Perhaps our formal proofs using induction won't need two cases!

Let us express the related concept of a well-founded set in our notation. Let S be a subset of U. An element y in S is called a *minimal element* of S if no element smaller than y (with respect to <) is in S; i.e. if

$$y \in S \wedge \forall(x\colon x<y\colon x \notin S)$$

$(U,<)$ is well-founded if every nonempty subset of U contains a minimum element, if for every subset S of U the following holds:

(3) **Well-foundedness**: $\neg empty(S) \equiv$
$\exists(y:: y \in S \wedge \forall(x\colon x<y\colon x \notin S))$

Later, we shall return to the notions of induction and well-foundedness and prove them equivalent.

Specifications in programming

One reason for using a formal notation is to be able to make precise and clear what is ambiguous or confusing. Consider the following statement: every value of $b.(i..j)$ that is not in $b.(h..k)$ is in $b.(h..k)$. (Here, b is an array and $b.(i..j)$ denotes the segment of b consisting of $b.i$ through $b.j$.) Such contorted and confusing statements do appear in informal specifications of programs. We can place this in our notation as follows:

(4) $\forall(v\colon v \in b.(i..j)\colon v \notin b.(h..k) \Rightarrow v \in b.(h..k))$

Of course, it may still seem confusing, but let us now simplify it:

(4)
$\equiv$ {Law of Implication}
$\forall(v:: \neg v \notin b.(h..k) \vee v \in b.(h..k))$
$\equiv$ $\forall(v:: v \in b.(h..k) \vee v \in b.(h..k))$
$\equiv$ $\forall(v:: v \in b.(h..k))$

Thus, the original statement is equivalent to the muc simpler 'every value in $b.(i..j)$ is also in $b.(h..k)$.' Fo the record, typically two thirds of the students shown thi problem are not able to deal correctly with (4) *unless an until* they apply the formal rules of manipulation.. Typically, the student says that (4) is equivalent to *false*.

Let me now specify an algorithm. The form and conten of a specification of an algorithm colors —indeed, pro vides the insight for— algorithmic development. So i makes sense to specify the algorithm precisely and simpl before beginning the development.

Consider an integer array $b.(0..n-1)$, where $n \geq 0$. Con sider any segment $b.(i..j-1)$ of b; we can compute th sum of its elements. We want an algorithm that finds th maximum such sum over all segments of b.

First define the sum of a segment:

$$S_{i,j} = \Sigma(k\colon i \leq k < j\colon b.k)$$

Next, give the result assertion R of the algorithm:

(5) $R\colon m = MAX(i,j\colon 0 \leq i \leq j \leq n\colon S_{i,j})$

Difficult this was not. Later, we will see how specifica tion (5) will guide the algorithmic development. But fc now, we can discuss one advantage of having written (5).

What is the maximum sum over all segments of the arra $b = (-1,-8,-4)$? Ask your students; if they depend o the informal specification, half will say -1 and the othe half 0, depending on whether they think of the empty seg ments as belonging to b. The formal specification make clear what is meant: empty segments are included, so th the answer is 0.

Note that $S_{i,j}$ does not include the value $b.j$. For our pu poses, for our formal manipulations, ranges like $i \leq k <$ are often preferred over ranges like $i \leq k \leq j$. Argumen concerning this have been put forth by Dijkstra [4], [5 Here is his argument for inclusion of the lower bound an exclusion of the upper bound:

> Exclusion of the lower bound, as well as inclusion c the upper bound, would have forced bounds outsid the realm of the natural numbers. I intend to let lower bound never exceed an upper bound; then w have the advantage that two ranges can be joined t form a single one if the upper bound of the one equal the lower bound of the other. Finally, the number c elements in the range equals the difference of th bounds. It now stands to reason to identify the ele ments of a sequence of length M —rows of a matrix c characters of a string— by a subscript in the rang

$0 \le subscript < M$.

Let me give one more example of a specification of a program, one that leads directly to the algorithm. Given is a nonempty sequence $b.(0..n-1)$ of integers that, lexicographically speaking, is not the largest (e.g. $(8,5,3,3)$ is the largest sequence that can be built with the bag $\{3,3,5,8\}$). Write an algorithm that changes the sequence into the next largest one using the same integers. This seems like a difficult algorithm until we specify more precisely what the next largest sequence is. First, let j be the index of the leftmost element of b to change. Since this element is to become larger and is replaced by something to its right, j is the largest value satisfying

$b.j < b.(j+1) \wedge$
$b.(j+1..n-1)$ is a non-increasing sequence.

or $\quad j = MAX(k:\ k < n-1:\ b.k < b.(k+1))$

Next, one must specify which value $b.i$ of $b.(j+1..n-1)$ is to be placed in $b.j$. Since the next largest permutation of b is to be created, $b.i$ should be the smallest value of $b.(j+1..n-1)$ that is greater than $b.j$. Since $b.(j+1..n-1)$ is non-increasing,

$$i = MAX(k:\ k < n:\ b.j < b.k).$$

By the definition of i, and since $b.(j+1..n-1)$ is non-increasing, the sequence $b.(j+1..i-1)\ \hat{}\ b.j\ \hat{}\ b.(i+1..n-1)$ is also non-increasing. Since the result desired is the smallest permutation of b that is larger than the initial b, we can specify the algorithm as implementing the assignment

$$b := b(0..j-1)\ \hat{}\ b.i\ \hat{}\ reverse(b.(j+1..i-1)\ \hat{}\ b.j\ \hat{}\ b.(i+1..n-1))$$

where j and i are defined above.

In this instance, refining or detailing the specification has led to a specification in which the algorithm to use is more apparent.

This ends our short excursion into the propositional and predicate calculi and notations to be used in formulas. We have attempted to show that formalizing informal statements can lead to simpler, clearer, and more precise statements, thus increasing our understanding. Further, the formal notations we use will be a factor in how simple our formalizations will be, so they must be chosen carefully. They should be geared to our manipulative needs, which means that they and the manipulation rules we use should be designed to minimize the text we must write.

The manipulation of formulas, of course, is aimed at proving properties of the objects being dealt with. Let us now turn to a discussion of such proofs.

3. The syntactic proof

We mentioned earlier the notion of a syntactic proof: sequence of symbolic transformations according to give rules. The notion of such a proof has been around fo some time, having been championed by David Hilbert one of the greatest mathematicians of all time. In fact Hilbert's program was to formalize *all* of mathematics a a set of axioms (or axiom schema) and inference rules.

Students may be taught about an axiom-inference-rul system, but rarely are they shown its real power; rarely i it used for anything interesting. Students are simply no taught to consider it a useful tool. More and more, I view the syntactic proof as a necessity, as a way of forcin myself to achieve a rigor and simplicity and understand ing that I would not otherwise achieve. Especially in pro gramming, with the myriad of details that have to b manipulated and understood, it essential to attempt t strive for the rigor and precision that the syntactic proo requires.

Some question the use of the syntactic proof, feeling tha it is cumbersome, complex, and difficult exactly in pro gramming, where there are so many details. It require symbol manipulation, and that is a task best left to th computer, for humans are bad at it. I disagree with thi viewpoint, and I generally find that the work needed t produce a syntactic proof can lead to simpler proofs an better understanding.

Let me quote Hilbert, from his famous lecture at th Second International Congress of Mathematicians in Pari in 1900, in which he outlined his famous 10 problems (2 in the manuscript, but only 10 mentioned in the lecture):

> It remains to discuss briefly what general requirement may be justly laid down for a solution of a mathemati cal problem. I should say first of all, this: that it b possible to establish the correctness of the solution b means of a finite number of steps based upon a finit number of hypotheses that are implied in the statemen of the problem and that must be exacty formulate This requirement of logical deduction by means of finite number of processes is simply the requiremen of rigor in reasoning. Indeed, the requirement of rigo which has become a byword in mathematic corresponds to a universal philosophical necessity o our understanding ... only by satisfying this require ment do the thought content and the suggestiveness o the problem attain their full value. ...
>
> It is an error to believe that rigor in the proof is th enemy of simplicity. On the contrary, we find it con firmed in numerous examples that the rigorous metho is at the same time the simpler and the more easil comprehended. The very effort for rigor forces us t discover simpler methods of proof. It also frequentl leads the way to methods that are more capable o

development than the old methods of less rigor. ...

Wherever mathematical ideas come up, ..., the problem arises for mathematicians to investigate the principles underlying these ideas and so to establish them upon a simple and complete system of axioms ... (Quoted from *Hilbert*, by Constance Reid, Springer Verlag, New York, 1983.)

During his time, Hilbert's program for formalizing mathematics received its share of criticism, with some mathematicians objecting to his 'reducing the science to a meaningless game played with meaningless marks on paper'. However, it is precisely the shuffling of meaningless symbols according to given rules that provides confidence! Couched in terms of English and informal mathematics, an argument may be difficult to understand and ambiguous. Once we agree on the formulation of the problem in a formal notation, then checking a well-written formal proof requires only checking that each rule was correctly applied.

Criticisms are similarly made about the use of formalism in programming. In the past, these criticisms were partially valid, in that we had difficulty ourselves in using the formalism ourselves. However, advances are such that it is clear from the literature that formalism can be used to great advantage in developing algorithms. At this point, it is only lack of education that is hindering progress.

Students —and anyone not experienced with symbol manipulation— have great difficulty at first in applying formal methods. Once they have made a textual substitution, say, they no longer 'understand what the formula means' and are hesitant to make further manipulations for fear of making mistakes. And because of this, they equate rigor with rigor mortis, with a stiffening of their abilities.

I have given courses in which I had actually to guide a student's hand as the student made its first textual substitution in connection with using the assignment statement axiom! This was depressing, for symbol manipulation is what programming is all about. Such courses, expected to be on the development of programs, had to spend far too much time on the predicate calculus and symbol manipulation. And these were not dumb or inexperienced students; they simply hadn't received a proper education.

The cure for this is a study of the use of logic as a tool in our work, with many examples chosen to illustrate the effectiveness of the approach; enthusiasm on the part of the instructor; a concentration on improving penmanship, for it is very important in reducing careless mistakes; and *lots of practice*.

Let me now turn to an extended, important, example of the syntactic proof.

Induction and well-foundedness again

We now turn to the proof of equivalence of mathematica induction over a partially ordered set $(U,<)$ and th well-foundedness of $(U,<)$. (I am indebted to Edsger W Dijkstra for the proof.) We have already given forma definitions of mathematical induction (2) and well foundedness (3). To prove their equivalence, we nee only show how to transform one into the other using sub stitution of equals for equals. This is done in Fig. 0 on th next page —what a simple proof!

The result itself is particularly noteworthy because i shows that mathematical induction has a formal basis induction can be applied whenever (and only then) th universe upon which it is to be applied is well-founded The result is completely general; one doesn't need a dif ferent argument to allow induction on natural numbers trees, lexically ordered pairs, lengths of derivations in grammer, etc; one need only know that the set under con sideration is well-founded.

Finally the result shows the student that mathematica induction has a firm basis; we know precisely the condi tions under which it can be used.

In light of all these advantages, it is disheartening to not that few of the current texts in discrete mathematics con tain *any* proof about when induction can be used, muc less this one. Most of the texts simply state the principl of mathematical induction as a *Grand Principle* that th student should believe and absorb without questioning it validity. And arguments concerning induction over othe sets are left to the imagination of the reader.

Interestingly enough, some computer scientists dislike thi syntactic proof, feeling that it does not convey the 'intui tion' behind induction. (Some also do not like formula tion (2) because it does not separate the base case fron the other cases, or do not like writing (2) as an equalit instead of as an implication; that is a different story.)

I asked one such colleague to give me *his* proof o equivalence; it is shown in Fig. 1 —I have changed nota tion and rewritten a bit, but not much. The proof relies o the finite-chain property, which we now explain. *decreasing chain* is a sequence $x_0, x_1, x_2, \ldots$ of element of U such that $\forall(i:\ 0\leq i:\ x_{i+1} < x_i)$. The finite-chai property states that all decreasing chains have finit length:

(6) **Finite-chain property**: $\forall(y :: \ DCF.y)$

where function DCF is defined as

$DCF.y$ = 'Every decreasing chain beginning with y is finite'

$\neg empty(S) \equiv \exists(y{::}\ y \in S \wedge \forall(x{:}\ x<y{:}\ x \notin S))$ (This is (3))

$\equiv$ {Complement both sides, use Law of Negation and DeMorgan's Law}
$empty(S) = \forall(y{::}\ y \notin S \vee \exists(x{:}\ x<y{:}\ x \in S))$

$\equiv$ {Define a predicate P: $P.x = (x \notin S)$ and replace occurrences of S by P}
$\forall(x{::}\ P.x) = \forall(y{::}\ P.y \vee \exists(x{:}\ x<y{:}\ \neg P.x))$ (This is (2))

Figure 0. Syntactic proof of induction and well-foundedness

Lemma. $(U,<)$ is well-founded iff the finite-chain property holds (see (6)).

Proof: The proof of this lemma is trivial.

Theorem. $(U,<)$ admits induction iff if it is well-founded.

Proof. We prove the two directions separately. First, assume that $(U,<)$ is well-founded. We shall prove that

(1.0) $\forall(y{::}\ P.y \vee \exists(x{:}\ x<y{:}\ \neg P.x)) \Rightarrow \forall(x{::}\ P.x)$

Let P be a predicate on U that satisfies the antecedent of the implication in (1.0). Consider the set S defined as $\{x \mid \neg P.x$
If we establish that S is empty, we shall have proved one direction of the theorem.

Suppose that S is non-empty. Since $(U,<)$ is well-founded, S has a minimal element u (say). We show a contradictic
which proves that the assumption that S is not empty is false. Since u is a minimal element of S, every element of U that
less than u cannot be in S; i.e. every such element satisfies P. Then, by our assumption about P, it follows that u satisfies
In other words, u is not in S, which is the desired contradition.

For the other direction, we assume that $(U,<)$ admits induction (i.e. (1.0) holds for all P) and show that it is well-founde
We assert (without formal proof) that the following is a tautology: either $DCF.y$ holds or there exists an x such that $x <$
and $\neg DCF.x$ holds:

(1.1) $\forall(y{::}\ DCF.y \vee \exists(x{:}\ x<y{:}\ \neg DCF.x))$

Using the induction principle, we conclude by mathematical induction that $\forall(x{::}\ DCF.x)$ holds. By the lemma, $(U,<)$
well-founded.

Figure 1. Alternative proof of induction and well-foundedness

My colleague's proof is given in Fig. 1, and I invite you to compare it with the proof in Fig. 0. Note that both use the trick of equating a property with the set of values that do not satisfy it. But one proof is five times the length of the other. One requires two separate proofs; the other doesn't. One requires the fact that a well-founded set satisfies the finite-chain property; the other doesn't. One requires a proof by contradiction; the other doesn't.

More time spent comparing proofs in this fashion would give the student an appreciation for simple proofs, as well as a sense of discrimination when reading others' proofs.

The proof that well-foundedness is equivalent to the finite-chain property is interesting in its own right. Fig. 2 contains a version of the proof given by Dijkstra in [5]. This proof is not completely formal, because the property $DCF.x$ has not been formalized. Formalizing DCF isn't necessary —and indeed would just introduce unnecessary clutter. It is this search for the right blend of formalism and informalism that makes mathematics, as well as programming, an art as well as a science.

We prove

(2.0) '$(U,<)$ well-founded' $\equiv \forall(y{::}\ DCF.y)$

Assume the lefthand side is true. Since $(U,<)$ is well
founded, it admits induction and the righthand side i
equivalent to

$\forall(y{::}\ DCF.y \vee \exists(x{:}\ x<y{:}\ \neg DCF.x)$

which is evidently true.

Now assume the lefthand side of (2.0) is false. The
there exists a nonempty set S with no minimal elemen
i.e.

$\forall(y{:}\ y \in S{:}\ \exists(x{:}\ x<y{:}\ x \in S))$

Hence, for all y, y in the nonempty set S, $\neg DCF.$
holds, and the righthand side of (2.0) is false.

Figure 2. Proof of equivalence of induction and the finite-chain property

4. Developing correct programs

This topic has received much attention in the past ten years or so. The methods seem worthwhile and are spreading, so I won't explain much here. I am speaking of the notions of developing a program hand-in-hand with its proof of correctness, which has its roots in Hoare's work [6], was created by Dijkstra in the middle 1970s [7], was further publicized by myself [8], and is becoming traditional enough for other texts to discuss it (e.g. [3]). Martin Rem's column on algorithms in *Science of Computer Programming* is a good place to turn to for more examples and discussions of the methods. Much of our requests for more rigor and for an agility with the predicate calculus stem from the kinds of formal manipulations one does when developing program and proof hand-in-hand.

The methods for developing proof and program hand-in-hand are usually attacked claiming that the amount of detail in a large program makes formalization infeasible: 'You *can't* expect us to prove every single subroutine and interface correct.'

The reply to this has several pieces. First, as Hoare has said, within every big program is a little program trying to get out, and the methods proposed often let this little program out. Thus, the big programs turn out to be smaller than we thought. Second, the attempts at formalization often lead to simplification, generalization, and better understanding, giving a cleaner product that reduces significantly the time required in debugging and validation. Therefore, extra time spent in the initial design and programming may well be saved at a later period of the project. Third, suitable use of abstraction will reduce the amount of detail that has to be considered at any one time. Fourth, it is not the case that *everything* must be formalized. Look at the proof in Fig. 2; the definition of $DCF.x$ has not been formalized —indeed, its formalization would have been counterproductive. In the same way, in a program and its proof we must learn to formalize exactly the right parts, and no more. It is this necessity to find the right balance between formality and precise informality that makes programming an art as well as a science.

Please don't misunderstand me; I am not saying that every large program can easily be proved correct. Indeed, we have little experience with the methods on large programs, and relatively few people apply formal proof methods in developing even small programs. And I am not saying that *I* have complete control over my own programming habits. Nevertheless, enough experience has been gained that those who know the method well feel that it belongs in the toolkit of every professional programmer.

I give here a partial development of an algorithm specified earlier because it illustrates so nicely the use of formal syntactic manipulation in developing and present ing an algorithm. If you have difficulty with it, ask your self whether the difficulty is with your unfamiliarity witl the methods and notation or with presentation itself.

Recall the specification of the program for finding th sum of maximum-sum segment of an array. Given is ai array $b.(0..n-1)$, where $n \geq 0$. Desired is an algorithn that stores a value in m to establish

$$R:\ m = MAX(i,j:\ 0 \leq i \leq j \leq n:\ S_{i,j})$$

where $S_{i,j}$ is defined by

$$S_{i,j} = \Sigma(k:\ i \leq k < j:\ b.k)$$

Assuming that a loop will be used, we develop a first cu $P0 \wedge P1$ at a loop invariant by replacing variable n in l by a fresh variable k:

$$P0:\ 0 \leq k \leq n$$
$$P1:\ m = MAX(i,j:\ 0 \leq i \leq j \leq k:\ S_{i,j})$$

Using the bound function $n-k$, we arrive at the loop

```
$k,m := 0,0$;
do $k \neq n \rightarrow$
    Establish $P1[k:=k+1]$;
    $k := k+1$
od
```

It remains to determine how to establish $P1[k:=k+1$ given $P0$, $P1$, and $k \neq n$. To do this, we rewrit $P1[k:=k+1]$ so that it has a term that resembles $P1$ using a range-splitting rule:

$$\begin{aligned}
& P1[k:=k+1] \\
\equiv\ & m = MAX(i,j:\ 0 \leq i \leq j \leq k+1:\ S_{i,j}) \\
\equiv\ & \{\text{Split the range } 0 \leq i \leq j \leq k+1\} \\
& m = MAX(i,j:\ 0 \leq i \leq j \leq k:\ S_{i,j}) \\
& \qquad max\ MAX(i,j:\ 0 \leq i \leq j = k+1:\ S_{i,j}) \\
\equiv\ & \{\text{Eliminate bound variable j in the second term} \\
& m = MAX(i,j:\ 0 \leq i \leq j \leq k:\ S_{i,j}) \\
& \qquad max\ MAX(i:\ 0 \leq i \leq k+1:\ S_{i,k+1})
\end{aligned}$$

Because $P1$ is initially true, this can be established by

$$m := m\ max\ MAX(i:\ 0 \leq i \leq k+1:\ S_{i,k+1}).$$

However, the second operand of the infix *max* operatc takes time $O(k)$ to evaluate, and we look for ways c strengthening the loop invariant to make this calculatio more efficient. Introduce a new variable c with definitioı

$$P2:\ c = MAX(i:\ 0 \leq i \leq k:\ S_{i,k})$$

The range of i is $0 \leq i \leq k$ instead of $0 \leq i \leq k+1$ so that r and c are defined in terms of the same segment of array l Some further just-as-simple calculations, which we leav to the reader, lead to the algorithm

```
$k,m,c$ :=0, 0, 0;
{$invariant$: $P0 \wedge P1 \wedge P2$}
do $k \neq n \rightarrow$
    Establish $(P1 \wedge P2)[k:=k+1]$:
        $c$ := $(c+b.k)$ $max$ 0;
        $m$ := $m$ $max$ $c$;
    $k$ := $k$+1
od
```

A beautiful, linear in n, algorithm results. Unfortunately, presenting this algorithm *always* has the difficulty that the audience is not familiar with the rules used to manipulate the formulas; they have not internalized them, and therefore cannot follow the presentation easily and cannot believe that someone could use such a method. Lack of education hinders their understanding.

5. The hazards of examples

An example can certainly be worthwhile, for instance if it is used as a redundant piece of information to help build the reader's confidence in his understanding of a concept, definition, theorem, algorithm, etc. And the younger the student (intellectually speaking) the more examples may be needed to build this confidence and expertise. However, too often the example is used as the major method of explanation, essentially as a crutch by the *writer* to eliminate the need (so (s)he thinks) for a clear, rigorous, explanation. Thus, an example is used as a substitute for the specification, or an example of 'stepping through' execution of a loop is given as the only explanation provided for a loop, or a complete algorithm is explained only in terms of an example.

A brief illustration will suffice to show the state of our textbooks. A major text in data structures, which is good in many ways, gives the following sentence as the *only* explanation of a certain algorithm: 'The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2, to convince himself that he understands the mechanism.'

This text was using a technique that I call 'programming by example': the reliance on an example (or two) for insight when developing or presenting an algorithm. This practice is less prevalent today than ten years ago, but it is still used far too often.

Let me give another illustration of this practice from the literature. Article [0] describes a neat algorithm for finding the minimum number of editing operations needed to change a given sequence of characters A into a given sequence of characters B. Two kinds of operations are allowed: delete a character from A and insert into A a character from B. The paraphrased description begins with (I have changed variable names):

Let $m.(i,j)$ be the edit distance between $A.(0..i-1$ and $B.(0..j-1)$. $m.(i,j)$ makes sense even when i o j is zero; .... These values are arranged as a matri with 1+#A rows and 1+#B columns. For example, i $A=abcabba$ and $B=cbabac$, the matrix of edit dis tances is

(7)
$$\begin{array}{ccccccc|c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & \\
1 & 2 & 3 & 2 & 3 & 4 & 5 & a \\
2 & 3 & 2 & 3 & 2 & 3 & 4 & b \\
3 & 2 & 3 & 4 & 3 & 4 & 3 & c \\
4 & 3 & 4 & 3 & 4 & 3 & 4 & a \\
5 & 4 & 3 & 4 & 3 & 4 & 5 & b \\
6 & 5 & 4 & 5 & 4 & 5 & 6 & b \\
7 & 6 & 5 & 4 & 5 & 4 & 5 & a \\
\hline
 & c & b & a & b & a & c &
\end{array}$$

In this example, the entry $m.5.4$, which lies at th intersection of rows 5 and column 4 (row and colum numbers start with 0) is the edit distance betwee *abcab* and *cbab*. The value in that position is because *abcab* can be tranformed into *cbab* by delet ing the leading a and c from the first string and the inserting a c at the front, but there is no shorter edi script for the transformation.

The paper proceeeds to describe properties of such edi matrices *m solely with reference to this example*. In fac the *whole description* of the algorithm is based exampl (7). Nowhere is there a proof of the properties of m; the must be inferred from (7). There is a 'proof of correct ness' of the algorithm at the end of the paper, but it i couched in vague terms, without a definition of m, and i opaque.

Can you conceive of a mathematics paper that deals wit concepts presented solely by example? Why do we hav to put up with it in computer science? When will edito and referees institute publication standards that eliminat this practice?

Actually, the definition of m and proofs of its propertie are rather simple. I maintain that the minimum numbe $m.r.c$ of editing operations to transform $A(0..r-1)$ int $B(0..c-1)$ is given by $m.i.j$ defined by

```
(8) $m.r.c$ =
    if $r=0 \rightarrow c$
     [] $c=0 \rightarrow r$
     [] $r>0 \wedge c>0 \wedge A(r-1)=B(c-1)$
            $\rightarrow m(r-1,c-1)$
     [] $r>0 \wedge c>0 \wedge A(r-1)\neq B(c-1)$
            $\rightarrow 1+min(m(r-1,c),m(r,c-1))$
    fi
```

Let us discuss this definition carefully; once we agree th it does indeed define the minimum number of editin operations we can forget about its interpretation and *wor* *solely with the definition*, thus placing the description of the algorithm on a rigorous and precise foundation. It is (almost) always better to translate the informal ideas as soon possible into a precise, formal form and thereafter to forget about the informal ideas and work solely with their formal definition.

The first two lines are obvious; to transform $A(0..-1)$ into $B(0..c-1)$ requires inserting the first c characters of B (that's c editing operations), and to transform $A(0..r-1)$ into $B(0..-1)$ requires deleting all characters of $A(0..r-1)$ (r editing operations).

Now consider $m(r,c)$ for $r,c \neq 0$. If $A(r-1)=B(c-1)$, transforming $A(0..r-1)$ into $B(0..c-1)$ is the same as transforming $A(0..r-2)$ into $B(0..c-2)$, as defined on the third line of definition (8). If $A(r-1) \neq B(c-1)$, then one can either

transform $A(0..r-2)$ into $B(0..c-1)$ and
  delete $A(r-1)$, or
transform $A(0..r-1)$ into $B(0..c-2)$ and
  append $B(c-1)$.

In either case, the number of editing steps is thus given as on the fourth line of definition (8).

One of the properties of m that the reader is supposed to glean from example (7) is that adjacent values differ by at most 1 and that diagonals are non-decreasing and increase by at most 2. The proof of this property, done using syntactic proof methods, gives far more confidence in the property than does example (7) (see [1]).

As mentioned above, a valid use of the example is to provide a redundancy so that the inexperienced reader can gain a measure of confidence in his understanding. However, the inexperienced reader is often likely to use the example as the major basis for understanding, and this is a dangerous practice. As an illustration of this, I offer the following.

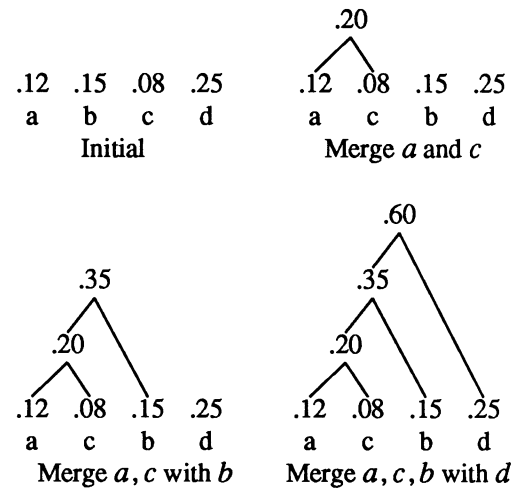
In a course on data structures, Huffman's algorithm for constructing a binary tree with certain properties from a list of (at least one) real numbers was being discussed. Consider each real number to be a tree with a single node whose value is the real number; the initial set S of trees is iteratively changed into a set containing only one tree by the algorithm below, given in my notation. I don't state precisely the task of the algorithm, except to build a tree, for that is not germane to this discussion.

Conventional set notation is used, with $\#S$ denoting the size of set S. A tree is considered to be a triple (*root value*, *left subtree*, *right subtree*), with *s.root* being used to refer to the first component of tree s —the value of the root node. Function *MIN.S* yields the tree in S that has the minimum root value.

```
do $\#S > 1$ $\rightarrow$ var $x := MIN.S$;  $S := S - \{x\}$;
                 var $y := MIN.S$;  $S := S - \{y\}$;
                 $S := S \cup \{(x.root+y.root, x, y)\}$
od .
```

Thus, at each iteration two trees are removed from S and
new tree (with the two removed trees as subtrees) i
inserted. The algorithm was not presented as concisely a
this in the text being used; it took *eleven* lines of Pascal
English.

In order to help the student understand the algorithm, a
example like the following was presented in the text:



Now, a not-inconsequential number of students looked a
the example rather than the algorithm and felt they under-
stood the algorithm, especially since the instructor use
exactly the same example while discussing the algorithr
in class. However, the example gave the wrong impres-
sion because it did not treat a general enough case. Give
the task of executing the algorithm on other data, thei
trees *always* had the shape shown in the above example (
right subtree is always a leaf), and this simply isn't th
case with this algorithm.

A combination of things led to this situation: the student
were too lazy or hadn't been taught to study the algorithr
itself, the text presented an unfortunate example, and th
instructor reinforced the lack of understanding by present-
ing the same example in class.

The lesson to be learned is that examples can indeed b
the cause of a problem. The student should be warne
about the dangers of examples and forced to study a
algorithm in terms of its proof and not in terms of exam-
ples of its execution sequences. Teach the student to us
the example as a redundant piece of information to len
assurance, but not as a replacement for a full understand-
ing by way of proof.

6. Further comments on notation

I have made various comments on notation throughout this lecture —on the use of ⊃ for implication, on notations for expressing quantification, and the like. I don't think it is realized strongly enough how the notation we use colors our thoughts and habits. Of course, we have heard that restricting oneself to older versions of Fortran severely restricts the possibilities of algorithmic expression, and we believe that a programmer should know several programming languages simply to expand his notational and conceptual horizons. Nevertheless, even the most innocuous-looking choice of notation can have severe consequences.

Last year, I had the chance to work with a computing scientist whose field was algorithms, a really top-notch person. When sketching segments of algorithms in Pascal, he would invariably use a repeat loop instead of a while loop, and almost as invariably there would be a mistake —the segment would not work properly for the empty segment of an array or some similar thing. After a while, I asked him why he continued to use the repeat loop when it so often led to errors and when its proof rule was so much more complicated than that of the while loop. He replied, rather sheepishly, that, yes, he knew the while loop was typically a better choice, but the while loop always required a **begin** and **end** while the repeat loop did not (since the keywords **repeat** and **until** act as delimiters of the loop). Furthermore, the prettyprinter that he used always put the keywords on different lines:

```
while B do
   begin
     body
   end ,
```

thus making his programs ever so much longer. And on a workstation screen his effectiveness in reading a program depended on how many lines he could see at one time.

Thus, a seemingly innocuous decision about syntax in Pascal, together with a rather stupid decision by the prettyprinter designer to make programs appear as long as possible, severely hindered this scientist's work.

It behooves us to make students aware of the impact on notation, right from the beginning.

7. Conclusions

I have touched on a number of topics that I believe computer science students should be learning but are not. These topics have their factual part (e.g. the predicate calculus, a calculus for the derivation of programs), but far more than a bunch of facts is involved. The student should acquire a sense of the use of method, notation, and proof, a sense of taste, and the ability to discriminate on technical grounds. This requires a different approach i our texts and in our teaching.

I don't mean to imply that computer scientists don't hav discrimination and taste. I do believe that they don't fee these qualities are as important as I do. And I do believ that they don't realize the effect their teaching practice have on students.

I also don't want to leave the impression that I feel I hav all the answers on the problem of teaching programmers Programming —and the teaching— is a difficult intellec tual task, and I feel I am just beginning to learn enough t do it well. Nevertheless, I hope that my arguments an examples will help persuade the field that change i needed.

8. Acknowledgements

My debt to Edsger W. Dijkstra and his ideas will be obvi ous to all who are familiar with his work; his influenc can be seen throughout this manuscript. Thanks go t Prakash Panangaden and Fred Schneider for many com ments on various drafts of this manuscript.

9. References

[0] Miller, W., and E.W. Myers. A file comparison algo rithm. *Software—Practice and Experience 15* (11 (November 1985), 1025-1040.

[1] Gries, D., and W. Burkhardt. A more rigorou description of an algorithm for finding the minimun edit distance between two sequences. Tech. Rpt August 1987, Computer Science Department, Cornel University.

[2] Dijkstra, E.W., and Feijen, W.H.J. *Een Methode va Programmeren*. Academic Service, Gravenhage, Th Netherlands, 1984. (Also translated into Germa under the title *Methodik des Preogrammieren* Addison-Wesley, Germany, 1985.

[3] Backhouse, R.C. *Program Construction and Verifi cation*. Prentice Hall International, London, 1986.

[4] Dijkstra, E.W. Largely on nomenclature. EWD 76 March 1981.

[5] Mathematical induction and computing science EWD819, April 1982.

[6] Hoare, C.A.R. An axiomatic basis for computer pro gramming. *CACM 12* (October 1969), 576-580, 583.

[7] Dijkstra, E.W. *A Discipline of Programming*. Pren tice Hall, New Jersey, 1976.

[8] Gries, D. *The Science of Programming*. Springe Verlag, New York, 1981.

[9] Constable, R., et al. *Implementing Mathematics wit the Nuprl Proof Development System*. Prentice Hal Englewood Cliffs, New Jersey, 1986.