An Experience Teaching Formal Methods in Discrete Mathematics

J. Stanley Warford
Pepperdine University
Malibu, CA 90263
warford@pepperdine.edu

Abstract

In spite of recent calls to incorporate formal methods into the computer science curriculum, the effort is still controversial and proceeding slowly. This paper presents our experience in restructuring the undergraduate Discrete Mathematics course to include formal methods. It concludes with some philosophical ruminations about the place of formal methods in the computer science curriculum in general.

Introduction

Six years ago Edsger Dijkstra addressed the ACM Computer Science Conference and called for the elevation of formal methods to a more prominent place in the discipline and curriculum of computer science. His talk was published in the Communications [3] along with comments from other scientists in the field [8]. Some agreed with Dijkstra's contention that computing is a "radical novelty" in need of a corresponding radical approach. Others took issue with his contention that formal methods should be taught as the primary tool for software development [12].

Two years later David Gries addressed the SIGCSE conference [5] with basically the same plea that formal methods needs to be taught as a core subject to undergraduates. He argued for teaching a calculational style of reasoning so that students will be as comfortable with proof techniques as they are with algebraic manipulations.

The calls for changes in the computer science curriculum have produced a debate on whether formal methods should be emphasized and, if so, then how [1, 2, 4, 7, 9]. An example of the influence of these ideas is contained in the Computing Curricula 1991 report [10]. The Appendix of the report contains several sample curricula identified as Breadth-First. The freshman course descriptions for these curricula all contain recommendations for a rigorous connection between specification and design, including loop invariants as an example.

In an effort to integrate formal methods into the curriculum we restructured the Discrete Mathematics course at Pepperdine University beginning the Fall Semester, 1993. The redesign of the course was motivated by our belief that the discrete math course, a required course during the first semester of the sophomore year, was the natural place in our curriculum to introduce the topic. The immediate impetus was the appearance of a new book by Gries and Schneider, *A Logical Approach to Discrete Math* [6], for which the authors had provided us a prepublication manuscript. This paper describes the course and lists a number of lessons learned from the experience. It concludes with some philosophical points on the place of formal methods in the curriculum.

Course description

The change in the course turned out to be a significant paradigm shift involving a trade-off of breadth for depth. The central idea behind the book [11] is to apply the logic machinery that has been developed to facilitate program proofs to topics that are traditionally taught in the Discrete

Mathematics course. The first half of the four-hour semester course is now devoted to an in-depth treatment of propositional and predicate calculus. The remainder of the course applies equational proof techniques to the traditional topics in discrete math. Table 1 shows the organization of the new course.

Table 1: Course topics

Weeks	Topic
0.50	Textual substitution, equality, assignment
0.75	Boolean expressions
1.00	Propositional calculus
0.50	Proof style
0.50	Review, exam
0.25	Solving word problems
1.00	Quantification
0.75	Predicate calculus
0.75	Predicates and programming
0.75	Review, exam
0.75	Theory of sets
1.25	Mathematical induction
1.00	Tuples, cross products, relations
0.75	Review, exam
1.00	Functions, partial orders
1.00	Modern algebra
1.00	Infinite sets
0.50	Review, exam

Topics from the old course that were sacrificed in this reorganization include logic networks, network minimization, coding theory, finite-state machines, and Turing machines. In most of these cases, the topic is covered in other courses in the curriculum. It was nice to have these topics treated in more than one course to serve as a reinforcement. Under the new organization, however, the linkage no longer is one of duplication but of application. For example, under the old curriculum a student would see logic networks and minimization in both Discrete Math and Computer Architecture. Now she sees it only in Architecture. Under the new curriculum, the student applies predicate calculus to problems in the following Data Structures course. We view this as a more efficient use of limited student contact time.

To emphasize the calculational approach all theorems are derived algebraically, and never by resorting to truth tables. Quantification of arithmetic and Boolean operators are presented consistently as shown in Table 2. In the same way that Σ is the quantified version of $+$, $\forall$ is the quantified version of $\wedge$.

Table 2: Quantification symbols

Operator	Quantified version
$+$	Σ
$\times$	Π
$\wedge$	$\forall$
$\vee$	$\exists$

Quantified expressions use a uniform notation for predicate and arithmetic operators as shown in Table 3.

Table 3: Notation

Conventional notation	Uniform notation
$\sum_{i=1}^{n} i^2$	$(\Sigma i \mid 1 \le i < n+1 : i^2)$
For the domain of positive integers $(\forall i)\ (P(i))$	$(\forall i \mid 0 < i : P.i)$

The real paradigm shift occurs when traditional discrete math topics like set theory and relations are all developed with the same formal methods machinery. For example, the set membership operator $\in$ is defined in terms of existential quantification by the axiom

$$F \in \{x \mid R : E\} \equiv (\exists x \mid R : F = E)$$

In English, the axiom states that F is an element of the set of expressions E over all x such that range R is true, if and only if there exists an x for which R is true such that F equals E. Similarly, set equality is defined in terms of universal quantification by the axiom

$$S = T \equiv (\forall x \mid : x \in S \equiv x \in T)$$

By the time we get to sets, students have had weeks of practice manipulating expressions like the quantifications on the right side of the above axioms. It is straightforward for them to use the definitions to prove properties of union, intersection, and subsets. Similarly, the development of cross products, relations, and functions employs formal methods of proof.

An example of the proof style is a proof of the theorem $S = \{x \mid x \in S : x\}$. Since this is a set equality, by the above axiom it suffices to prove that $v \in S \equiv v \in \{x \mid x \in S : x\}$ for arbitrary v.

$$
\begin{aligned}
& v \in \{x \mid x \in S : x\} \\
= & \quad \langle \text{Definition of set membership} \rangle \\
& (\exists x \mid x \in S : v = x) \\
= & \quad \langle \text{A previously learned trading theorem} \rangle \\
& (\exists x \mid x = v : x \in S) \\
= & \quad \langle \text{The previously learned one-point rule} \rangle \\
& v \in S
\end{aligned}
$$

Observations

The following observations are based on my experience with the new approach.

Proofs—The entire nature of the course changed. Rather than most exercises being based on examples and a few on proofs, now most exercises are proofs and only a few are based on examples. Proofs, which students used to dread, are now the favorite part. I believe this phenomenon can be traced to the detailed exposition of formal methods at the beginning. Students are used to algebraic manipulations from high school algebra, and learning the symbolic manipulations required of formal methods came quite naturally. I also found that proofs were easier to grade. With the old approach, I never knew if a student understood the steps in the proof but was having trouble with the English or if he did not understand the proof. Now that proofs have an explicit syntax and semantics it is easier to pinpoint a problem and evaluate a proof attempt for giving partial credit.

Unified treatment—It is intellectually satisfying to teach discrete mathematics with a unified approach. The discrete math course takes on the same nature as the traditional calculus course. In the same way that a calculus student learns in a linear fashion the manipulations necessary for taking derivatives and then integrals, the discrete math student learns in a linear fashion the manipulations necessary for formal proof and then its application to topics in program correctness and discrete mathematics. Formal logic is the glue that binds the topics together.

Notation—The notation shown in Table 3 takes some getting used to, but is to be preferred to the traditional notation. The fact that arithmetic and predicate operators can share the same syntactical notation when quantified makes calculations much easier to do and to teach.

An example of one benefit of this notation is seen in the expression

$$(\forall x) \ (\ (\exists y) \ P(x,y) \wedge Q(x,y))$$

written traditionally. Without precedence rules it is not obvious what the scope of $\exists$ is. Does it extend to Q or not? Unlike the older style, the parentheses in the quantified expression

$$(\forall x \mid : (\exists y \mid : P(x,y)) \wedge Q(x,y))$$

always indicates the scope making it clear whether the y in $Q(x,y)$ is a free variable.

Following courses—I found an immediate benefit the following semester in my Data Structures course. Knowing the quantification rules for summation helped in the derivation of statement execution counts in the analysis of algorithms. Knowing the formal properties of Hoare triples and loop invariants gave students and teacher a common vocabulary with which to explain and understand complex algorithms.

Justification—Students have a peculiar habit of questioning why they must learn the particular

topics we select for them. In particular, teaching formal methods for the sole purpose of program proof or program derivation can be difficult to justify. It is easier for students to accept formal methods when it is cast as a mathematical discipline that applies to discrete topics as well as programming topics. This broadening of the domain of application that the text achieves is a major benefit to the curriculum.

Philosophical ruminations

The utility of formal methods in computer science is still controversial. Some argue that formal methods are too difficult to teach to undergraduates. My experience with *A Logical Approach to Discrete Math* convinced me that formal methods are easily mastered at the undergraduate level. A surprising observation is that the approach seemed to help the weaker students more than the stronger ones. It seems that strong students will learn in spite of our approach. Rather than overwhelming the weaker students, formal methods gave them confidence in their newly learned skill of symbol manipulation.

Others argue that formal methods are too cumbersome to solve large software problems, and that we should therefore not spend time teaching them to our students. While it may be true that the benefits of formal methods have been oversold and that they are not the silver bullet for solving the software crisis, I found that the benefit of teaching formal methods in the discrete math course had immediate benefit in following courses.

One problem is that some advocates of formal methods either make or imply some rather extravagant claims. For example, Dijkstra [3] goes so far as to call Software Engineering "The Doomed Discipline" because it cannot even approach its goal. He disparages programming tools for software production and algorithmic visualization tools for education in favor of predicate calculus applied in a calculational style.

The problem with extravagant claims is that they frequently polarize their audiences. Many computer scientists disagree with promises made by formal methods advocates [8, 12]. Some people interpret the current state of affairs in computer science as evidence that formal methods have failed. In questioning the foundations of computer science, Wulf [13] states that he "... gave up on formal specifications and program verification" because "... the problem proved to be hard, and there was an easier problem at hand." He makes the point that the lack of a solution has so far been masked by advancing technology.

After all, formal methods have been around a long time. Where is the large commercial software project whose success can be demonstrably tied to the formal methods practiced by its developers? It is no wonder that excessive claims by the advocates of formal methods fail to rally the troops.

I fear, however, that we may be in danger of throwing out the baby with the bath water. I came away from this course convinced of two things. First, formal methods will not be the silver bullet for solving the software crisis in our industry. It may be one of several bullets. Second, we educators should, nevertheless, require a firm foundation of formal methods of all our undergraduate computer science students.

Practicality is not the issue. The issue is the philosophical question of what constitutes an education in a scientific discipline. An economics major who takes a calculus course may be satisfied by learning the rules for differentiating and integrating a few functions that are representative of her discipline. These simple skills are sufficient for her needs as a non mathematics major. However, no mathematics program would fail to teach the proofs of the fundamental theorems of its discipline to its majors.

Similarly, programming is an activity that almost anyone can practice with a minimal amount of learning. But a computer science major ought to be conversant with the theoretical foundation of programming, which is probably the most common activity of our discipline. The existence of

theory and its relationship to practice is what distinguishes a scientific discipline from a craft and a college from a trade school.

Because the theory exists, it belongs in the core of our curriculum. If a deeper understanding of the basis of computer science also has the effect of enabling our students to write better code, so much the better. I personally think that this side effect is inevitable.

Based on my experience with this course, I believe that the appropriate place for integrating formal methods in the undergraduate curriculum is the traditional discrete mathematics course at the sophomore level. Rather than the usual cursory introduction to the predicate calculus, the course should introduce the topic early and in enough depth to enable students to develop computational skills in proofs of mathematical theorems and program correctness. Such a reorganization of the discrete course means that other topics may need to be de-emphasized.

Conclusion

Formal methods deserves a more prominent place in the core undergraduate computer science curriculum. The reason for this assertion is not that formal methods may be a valuable software engineering tool, but that they are the fundamental theoretical foundation of programming.

References

[1] Berens, T., et al, ACM Forum. *Commun. ACM* 34, 9 (Sept. 1991), 16–18, 91–95.

[2] Denman, R., et al, Derivation of programs for freshmen. *ACM SIGCSE Bulletin* 26, 1 (March 1994), 116–120.

[3] Dijkstra, E.W. On the cruelty of really teaching computing science. *Commun. ACM* 32, 12 (Dec. 1989), 1398–1404.

[4] Fekete, A. Reasoning about programs: Integrating verification and analysis of algorithms into the introductory programming course. *ACM SIGCSE Bulletin* 25, 1 (March 1993), 198–202.

[5] Gries, D. Calculation and discrimination: A more effective curriculum. *Commun. ACM* 34, 3 (March 1991), 45–55.

[6] Gries, D. and Schneider, F.B. *A Logical Approach to Discrete Math* Springer-Verlag, New York, 1993.

[7] Lau, K.K., et al, Towards an introductory formal programming course. *ACM SIGCSE Bulletin* 26, 1 (March 1994), 121–125.

[8] Parnas, D.A., et al, Colleagues respond to Dijkstra's comments. *Commun. ACM* 32, 12 (Dec. 1989), 1405–1414.

[9] Saiedian, H. Towards more formalism in software engineering education. *ACM SIGCSE Bulletin* 25, 1 (March 1993), 193–197.

[10] Tucker, A.B., et al *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force* ACM Press, New York, 1991.

[11] Warford, J.S. Review number 9407-0412. *Computing Reviews* 35, 7 (July 1994), 340–341.

[12] Wielgus, W., et al, ACM Forum. *Commun. ACM* 33, 4 (April 1990), 396–398.

[13] Wulf, W.A. Is CS built on a foundation of sand? *Computing Research News* 4, 3 (May 1992), 2.

********ADT's References From Page 59********

References:

[1]. Gersting, Judith L., "A Software Engineering Frosting on a Traditional CS-1 Course", Proc of the 25th SIGCSE Technical Symposium on Computing Science Education, 233-237.

[2]. Jarc, Duane M., "Data Structures:A Unified View", SIGCSE Bulletin, Vol 26, No. 2, June 1994, 2-4.

[3]. Turner, A. Joe, "A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report Computing Curricula 1991", Comm of the ACM, Vol 34, No. 6, June 1991, 69-84.