The Case for Teaching Functional Programming in Discrete Math

Thomas VanDrunen

Wheaton College, Wheaton, IL

Thomas.VanDrunen@wheaton.edu

Abstract

Functional programming is losing its place in undergraduate computer science curricula, in part because of the attention given to many new ideas in the field. Nevertheless, undergraduates benefit from an early experience in a second programming paradigm, especially functional programming. The solution, advocated here, is to weave functional programming into the discrete mathematics course. Not only does this give a convenient, early occasion to teach functional programming, but it also allows the functional programming topics and discrete mathematics topics to illuminate each other. Furthermore, it provides a service course to students in mathematics and other majors.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education—Computer and Information Science Education; G.2 [*Discrete mathematics*]: General

General Terms Languages

Keywords Discrete mathematics, functional programming

1. Introduction

There is a wide variety of pedagogical and curricular approaches to introducing the field of computer science. This can be seen by perusing the available textbooks for CS 1 or similar courses. Even though the core topics for undergraduate programs in computer science have been standardized in such reports as the IEEE and ACM Computing Curricula reports [13, 14], there remain considerable differences across programs regarding what first programming language or paradigm works best, the relative emphases for theory, systems, and software development, the ordering and pairing of topics throughout the curriculum, and which new trends or advanced topics to integrate into the curriculum and how quickly.

In recent curricular trends, the topic of functional programming is being squeezed out of many undergraduate programs. In this paper we advocate retaining it as a core and early topic in the curriculum, specifically that the ideal place for it is in a freshman- or sophomore-level course in discrete mathematics. In Section 2 we make a brief defense for keeping (or reintroducing) functional programming in the undergraduate curriculum and observe the curricular pressures that make this difficult. Section 3 lays out the case for putting functional programming and discrete mathematics in the same course, showing that not only do the mathematical and programming content streams complement each other for computer science majors, but that they also together provide a useful service course for math majors and others. Precedents for this approach are briefly surveyed in Section 4. Section 5 addresses pragmatic aspects of implementing a course in discrete mathematics and functional programming, suggesting a course outline and reporting on the author's own experience. In Section 6 we conclude by reviewing the advantage of placing such a course in the curriculum.

2. Functional programming in the curriculum

2.1 The need for functional programming

Functional programming is vital to a well-rounded understanding of the field of computer science. Not only is the functional programming model of computation foundational to some approaches to the theory of computation, but important applications have been produced in it. Even if students end up programming primarily in an imperative or object-oriented setting, seeing another paradigm increases their understanding of the paradigm they use most. Moreover, many idioms and design patterns for object-oriented programming are essentially ways to import features or programming idioms from functional programming [7, 9, 12].

The SIGPLAN Workshop on Undergraduate Programming Language Curricula, meeting ahead of the 2008 revi-

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0942-4/11/10. . . $10.00

sions to Computing Curricula 2001, recommended a prominent place for functional programming:

> Shifting to functional programming is not merely about a change in syntax; rather, it forces students to approach problems in a novel way. This change increases their mental agility and prepares them for a life of practice in a world where languages continually grow, morph, and sometimes shift their perspective [2].

2.2 The squeeze on functional programming

The prevalence of functional programming surged in the 1980s with the publication of the original edition of *Structure and Interpretation of Computer Programs* [1], and many schools began using a dialect of Lisp as the first or primary programming language in their curriculum. Felleisen et al describe this popularity as short-lived because of the difficulty of then-available introductory material on functional programming and because of how greatly functional languages differ from dominant programming languages [6]. (They also propose a solution: Begin the curriculum with a functional language but follow it quickly with an object-oriented language. This way the students learn early how to think in both paradigms.)

With functional programming taken out of the introductory sequence, a typical student would be introduced to it in a sophomore- or junior-level programming languages course. This is where the IEEE and ACM Computing Curricula report (2001) places the topic (PL7) [14]. By that time, students have acquired the imperative or object-oriented paradigm (or a combination of them) as their native way of thinking about computation, and functional programming is viewed as a alternative curiosity.

Even that presence of functional programming in typical computer science programs is in jeopardy. Mark Bailey observes that as the field of computer science grows, new topics crowd for attention in a curriculum, and a core programming languages course is likely to absorb a lot of the pressure. "The emergence of offerings, at the undergraduate level, of courses in bioinformatics, wireless networking, security, game programming, robotics, and mobile computing. . . compete for a limited number of course slots in an undergraduate computer science major. . . . Liberal arts colleges, in particular, have greater curricular pressures" because of limited time alloted to major courses [3].

For functional programming in particular, we note that Computing Curricula 2001 does not place PL7 as a core topic—in fact, one of the main goals of the Workshop on Undergraduate Programming Language Curricula for the 2008 update to the Computing Curricula report was "to make the functional programming unit (FP), PL7 in the curriculum, required rather than elective" [2]. Although the 2008 update does require exposure to more than one programming paradigm, the CC 2008 Review Taskforce decided not to require functional programming in particular [13].

2.3 Other curricular needs to address

Some schools prefer to have an early course on the foundations of computer science that gives computer science majors the mathematical background to reason about computational models and their limitations and to appreciate the contributions of the theory of computation to computer science and to science in general. For example, the computer science program at Purdue University requires CS 18200, a freshman course that begins with discrete math topics and goes on to cover analysis of algorithms, proofs, automata, and computability. A course like this may also be of interest to students outside the computer science program.

Many computer science departments must provide programming or other computing courses to serve the needs of other programs at the school—most notably mathematics, engineering, and natural science. There is great advantage to giving the introductory courses for computer science majors double duty so that they also fill this need for service courses. Not only can they be used for recruiting (so that a few mathematics or engineering students can be "converted" to computer science), but for small departments it is a matter of efficient use of faculty load. Thus for many schools it makes sense for introductory computer science courses to be designed for other populations in addition to computer science majors.

3. Where should functional programming go?

In light of the need for functional programming in the computer science curriculum and its precarious position in either the introductory programming course or in a mid-level programming languages course, the question arises: Where should functional programming go in the computer science curriculum? We propose an answer: Put it in the discrete mathematics course. Call it, "Discrete Mathematics and Functional Programming," DMFP. Students can take it some time in their first three semesters *in parallel with* a traditional introduction to programming sequence.

Functional programming and discrete math are closely tied. It is immediately apparent that the two areas of study are related. To begin, functions and recursion are fundamental to both [11]. Moreover, proof by induction is an important topic of a discrete math course, which is also closely tied to recursion.

Functional programming illuminates discrete mathematics. The inclusion of functional programming in a discrete math course has a pedagogical benefit to the teaching of the discrete math topics. As Cong-Cong Xing has pointed out, the treatment of functions in discrete math differs enough from students' experience with functions in pre-calculus to cause a good deal of confusion. In particular, students have trouble differentiating between the name of a function and the function itself, and, along the same lines, trouble understanding functions as discrete objects or the idea of a higher-order function. A functional programming language gives a context in which to illustrate all these things [17].

Moreover, functional programming languages are good contexts for modelling other ideas in discrete mathematics. Types and lists each model sets, though in different ways. Lists also model sequences. Tuples model the elements of a Cartesian product. Relations can be modeled in many ways: lists of tuples, predicates, and matrices, to name three. Functions, of course, represent functions. Quantification can be illustrated algorithmically in functions that process lists.

DMFP puts functional programming early. It has become standard for students to take a course in discrete mathematics or discrete structures in their first or second year. The CC01 gives "Discrete Structures" 43 core hours, more than any other area. Putting functional programming here is a compromise between a functional-first approach and delaying functional programming until a later point; it incorporates the advantages of functional-first but still allows the introductory programming sequence to be taught in an object-oriented or imperative paradigm.

Specifically, by putting functional programming here, students will have experience in functional programming ideally in their first year, in their second year at the latest. This way they can digest it before a sophomore- or junior-level software development course and thereby appreciate the influence of functional programming on design patterns. Likewise, the students' experience in a second paradigm allows a later course in programming languages to be more advanced.

DMFP allows a thorough investigation of functional programming. If functional programming is not taught until a mid-level programming languages course, then that course likely spends only one unit on functional programming in a series of units on different programming paradigms. Students are unlikely to acquire a thorough understanding of, much less competency in, functional programming with merely a passing exposure. Functional programming would not get a full course's worth of attention in a DMFP course either, since it would be only part of the content stream. However, with functional programming topics being spread throughout the semester—and, more importantly, with students continually practicing functional programming—the students will gain and retain a more thorough skill base in the paradigm.

DMFP motivates discrete mathematics for computer science majors. Despite computer science and mathematics being kindred fields, computer science major populations include many math-averse students. Many are frustrated at the math requirements of the program and are slow to understand the relevance. The situation is more likely to be aggravated than remedied when the discrete math course is taught by the mathematics department. The functional programming component in a DMFP course provides a set of enjoyable topics and assignments to keep the computer science majors engaged. More importantly, the links between the mathematical topics and the pragmatics of programming are made explicit.

DMFP motivates programming for math majors. Computing is a vital topic for contemporary math majors. Many will need some level of competency in programming at some point in their studies, whether in professional practice as actuaries, to introduce algorithmic topics as high school math teachers, or in research as graduate students. Accordingly, math major programs typically include at least one semester of programming. Unfortunately, many math majors find programming to be foreign and become frustrated when they do not see any immediate relevance for their mathematical studies. Their frustration is especially understandable if they are dropped into a Java programming course that was not designed for their needs.

What better context to introduce programming to math majors than a math course? Just as the programming topics illuminate and motivate the math topics for computer science majors, the math topics do the same to programming topics for math majors.

The course easily can be designed to hit a sweet spot for both majors. There is a synergistic effect in bringing both student populations together. The slogan used in advertising the course taught by the author is *Computer science majors should learn to write proofs and math majors should learn to write programs* ***together***. Computer science majors who have had a previous or concurrent programming course will be at an advantage for learning functional programming, whereas math majors usually are better prepared for the course's proof-writing content. The two populations can partner and help each other.

DMFP provides a framework for talking about computer science foundations. Many computer science programs have an early course on the foundations of computer science that can give an overview of the field, especially of the theoretical aspects. Topics can include models of computation, automata, asymptotic growth of functions, correctness proofs, P vs NP, and the limits of computation. It is hardly possible to deal with these topics rigorously in an introductory course—in fact, for many programs these ideas appear in an elective if anywhere. However, touching on these ideas informally at the front of the curriculum allows the students to connect them with the practical material they see throughout the course of study.

A course in discrete mathematics provides the right framework for discussing these ideas, since they rely on sets, relations, functions, and graphs, not to mention logic and quantification. Moreover, even a smattering of functional programming provides simple illustrations of the computational issues and provides exercises accessible even to beginning students.

4. Precedents for this approach

Although the case for teaching functional programming and discrete mathematics together is here presented as a proposal—and it is not common practice—it is not altogether without precedent either. Roger Wainwright [15], Peter Henderson [8], Christelle Scharff and Andrew Wildenberg [11], and Cong-Cong Xing [17] have reported experiences in teaching Standard ML or Miranda in a discrete math course at five institutions for two decades. Moreover, the *Model Curriculum for a Liberal Arts Degree in Computer Science* by the Liberal Arts Computer Science (LACS) Consortium, which was produced in 2007 in response to Computing Curricula 2001, puts a course titled "Discrete Structures and Functional Programming" in the introductory sequence [4].

While many texts exist either on discrete mathematics or functional programming, not many are available for teaching the two together. Doets and van Eijck [5] and O'Donnell, Hall, and Page [10] have texts that use Haskell to illustrate discrete mathematics. Fenton and Dubinsky have a text that similarly uses ISETL [16]. Moreover, the author of this article has a forthcoming text that completely integrates discrete mathematics and functional programming, using Standard ML.

5. Implementing Discrete Mathematics and Functional Programming

In this section we list some suggestions in implementing the approach advocated in this paper. We provide a sketch of an outline for such a course and report on our experience in teaching a course that follows this program.

5.1 Course outline

Discrete mathematics courses, even those intended for computer science students, vary greatly in what topics are covered and in what order. Should the course start with symbolic logic since logic is the most fundamental science, or should "raw material" like sets or integers be introduced first? Should the students' first experience with proofs be on sets or integers, or a combination of the two? Should the chapters on relations and functions be ordered from general to specific (relations first, then functions) or specific to general (functions first, then relations)? Should graphs and relations be introduced together or separately? Should induction be introduced early as a basic proving tool or late as an advanced proof technique?

We believe that the DMFP approach is not tied to any particular ordering of the material. In the list below we describe the points of contact between the discrete math topics and the functional programming topics, but there are few hard dependencies between items in the list.

Sets. Functional programming makes heavy use of lists, and even though lists are ordered and sets are unordered, it is natural to introduce lists along side of sets as a way to model them in a programming language. Functions on lists model operations on sets. Sets and types also illuminate each other, and this point in the course is an opportune time to introduce simple user-defined types, such as SML's `datatype` construct (`type` in F# and OCaml). Tuple types are introduced to illustrate Cartesian products.

Symbolic logic. Boolean values and operations naturally model ideas from symbolic logic. Functions that return boolean values represent predicates. Whether the course begins with sets or with logic, once the students have seen both concepts, then multiple quantification can be used in and illustrated by algorithms. For example, an exercise asking the students to write a function that takes a list of integers and determines whether it contains an item that is the divisor of all the others requires the students to nest a universally quantified question inside of an existentially quantified question.

Proofs. Instructors may find that the portion of the course that introduces proofs has the fewest natural ties to programming. With a little effort, however, the course can show students how algorithms give insights into theorems and their proofs and that mathematical results and their proofs sometimes provide algorithms. As an example of the former, students can write a function that computes a powerset of a set and then experiment with sets of several sizes. The observation that the size of the power set doubles as the size of the original set grows leads to the theorem $|\mathscr{P}(A)| = 2^{|A|}$. Moreover, the algorithm itself suggests an outline for the proof. In the other direction, the Euclidean algorithm and the division algorithm grow naturally out a theorem about greatest common divisors and the quotient-remainder theorem, respectively.

Induction. Induction is one of the clearest places in the semester where functional programming ties in. If induction is introduced later in the semester, then by that time students already have experience thinking recursively, so proofs by induction will not seem as foreign to the students as is often the case. Structural induction and recursively defined types should be taught together, and students will find that proofs of structural induction will look very similar to functions on recursive types. Moreover, if structural induction is taught before mathematical induction, then an implementation of whole numbers using the Peano axioms makes the transition from structural induction to mathematical induction seamless. Moreover, proofs of algorithm correctness give a concrete motivation for mathematical induction.

Relations. Relations provide an opportunity to illustrate the trade-offs between different ways to represent or store information. Relations can be represented on a computer in at least three ways: as predicates (functions), as sets of pairs (lists of tuples), or as matrices. The students' experience in programming earlier in the semester also makes it easier to talk about applications of relations to different areas of computing, such as databases.

Functions. One goal of studying functions in a discrete math course is for the students to understand functions as mathematical objects. This is the time in the semester to talk about functions as first class values. This opens the way for introducing several idioms in functional programming, some of which have convenient ties to ideas in the study of functions. For example, the use of map is an illustration of the image of a function.

Cardinality and computability. Some of the deeper results of set theory provide the background for big ideas in the theory of computation. For example, Russell's paradox (there can be no set of all sets), countability (when cardinality is extended to infinite sets, reals are a higher order infinity than integers), and computability (the Halting Problem is not computable in known models of computation) form a triad of special topics.

Graphs. In a DMFP course, the section on graphs presents both the theoretical aspects of graph theory (the handshake theorem, isomorphisms) and the practical side (algorithms for searches, spanning trees, and shortest paths). The course could also explore the trade-offs of different ways to represent graphs on a computer.

The course should also include excursions into larger programming examples that exercise students' functional programming skills, illuminate the mathematical topics, and generally excite the students about the field. Obvious examples include a system for parsing and transforming text, an automatic theorem prover, and applications in game theory.

The combined experience in programming and discrete math provides the foundation for other and more advanced topics, especially if the course is spread through two semesters. Many discrete math courses have units on number theory, combinatorics, and discrete probability, and some explore mathematical structures such as boolean algebras, lattices, and groups. In any of these cases, the students' experience in programing allows for the inclusion of applications, uses, and illustrations in computing. If the course should include computer science topics such as a survey of automata and formal languages or complexity classes and asymptotic notation, students will have the mathematical background to deal with those ideas rigorously.

5.2 Experience

The author has taught discrete math with a functional programming component (using Standard ML) eight times. In six of those offerings the programming component has made up at least 40% of the course content and the two topic streams were fully integrated. The course is populated by both math majors and computer science majors, as well as a few students from other programs. The course is required for computer science majors; for math majors, the course is one of four options available for fulfilling a supporting requirement in computing. Students from other programs take the course as pure elective. The computer science majors are freshmen and sophomores, taking the course in parallel with a first, second, or third semester of programming in Java or C. There has been at least one case of a computer science major taking the course as her first course in the program. The math major population in the course is drawn from all class years; almost all of them have little or no programming experience coming into the course.

The math majors naturally are at an advantage over the computer science majors on the mathematical topics, and likewise computer science majors over the math majors on programming topics. However, neither population dominates the other in overall performance over the course of the semester, and top students in the class have come from either major.

Since the course is taught at a small college, the author can observe the long-term result of the course among computer science students by seeing how they put the ideas to use in later courses in the curriculum. In particular, the early exposure to functional programming has made students better prepared for topics in software development, programming languages, and analysis of algorithms. Moreover, the author has observed students expressing long-lasting appreciation for the ML language and for the usefulness of discrete mathematics.

It is harder to observe the long-term benefits of the course among math majors. However, enrollment in the course for the fall 2010 was up 275% over enrollment in fall 2009. (The course is offered only in the fall.) This did not correspond to an increase in enrollment in the computer science program but came primarily from an increase of math majors and non-majors signing up for the course. Among math majors the increased interest appears in part to be from math majors being advised by upperclassmen and mathematics professors that the course is good preparation for modern algebra (a change in the course meeting time also made it more accessible to math majors than in previous offerings).

6. Conclusions

Computer science, as a quickly-changing and still relatively young field, has resisted standardization of undergraduate curriculum and has seen many pedagogical trends come and go. It is not our present intention to advocate yet another trend but to make the case for a solution to a specific problem in light of the current curricular landscape.

Teaching a course in discrete mathematics and functional programming has the advantage of finding a place for functional programming in an increasingly full curriculum, showing the students a second or third programming paradigm early in their course of study, and making explicit links between ostensibly disparate topics. The approach is particularly appropriate for liberal arts colleges that value integration between fields and where the course can serve several populations of the student body.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw Hill and the MIT Press, Cambridge, MA, second edition, 1996.

[2] E. Allen, R. Bodik, K. Bruce, K. Fisher, S. Freund, R. Harper, C. Krintz, S. Krishnamurthi, J. Larus, D. Lea, G. Leavens, L. Pollock, S. Reges, M. Rinard, M. Sheldon, F. Turbak, and M. Wand. SIGPLAN programming language curriculum workshop: Discussion summaries and recommendations. *SIGPLAN Not.*, 43(11):6–29, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1480828.1480831.

[3] M. W. Bailey. Injecting programming language concepts throughout the curriculum: An inclusive strategy. *SIGPLAN Notices*, 43(11):36–38, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1480828.1480835.

[4] L. A. C. S. Consortium. A 2007 model curriculum for a liberal arts degree in computer science. *Journal on Educational Resources in Computing*, 7(2):2, 2007. ISSN 1531-4278. doi: http://doi.acm.org/10.1145/1240200.1240202.

[5] K. Doets and J. vanEijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing. King's College Publications, London, 2004.

[6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796804005076.

[7] P. Graham. Revenge of the nerds, May 2002. http://www.paulgraham.com/icad.html.

[8] P. B. Henderson. Functional and declarative languages for learning discrete mathematics. In *The Proceedings of the International Workshop on Functional and Declarative Programming in Education*, 2002. Technical Report No 0210 of the University of Kiel.

[9] P. Norvig. Design patterns in dynamic languages, March 1998. http://norvig.com/design-patterns/.

[10] J. O'Donnell, C. Hall, and R. Page. *Discrete Mathematics Using a Computer*. Springer, London, second edition, 2006.

[11] C. Scharff and A. Wildenberg. Teaching discrete strcutres with SML. In *The Proceedings of the International Workshop on Functional and Declarative Programming in Education*, 2002. Technical Report No 0210 of the University of Kiel.

[12] J. Spolsky. Can your programming language do this? In *Joel on Software* blog, August 2006. http://www.joelonsoftware.com/items/2006/08/01.html.

[13] The Interim Review Task Force. Computer science cirriculum 2008: An interim revision of cs 2001, December 2008. http://www.acm.org//education/curricula/-ComputerScience2008.pdf.

[14] The Joint Task Force on Computing Curricula. Computing curricula 2001, December 2001. http://www.acm.org/-education/education/education/curric_vols/cc2001.pdf.

[15] R. L. Wainwright. Introducing functional programming in discrete mathematics. In *SIGCSE '92: Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 147–152, New York, NY, USA, 1992. ACM. ISBN 0-89791-468-6. doi: http://doi.acm.org/10.1145/134510.134540.

[16] E. D. William E Fenton. *Introduction to Discrete Mathematics with ISETL*. Springer, London, 1996.

[17] C.-C. Xing. Enhancing the teaching and learning of functions through functional programming in ml. *Journal of Computer Sciences in Colleges*, 23(4):97–104, April 2008.