Functional and Declarative Languages for Learning Discrete Mathematics

Peter B. Henderson[1]

Butler University, Indianapolis IN, 46208, USA
phenders@butler.edu

Abstract. Discrete mathematics and logic are important foundations for the education of computer scientist and software engineers. Often this material is not introduced sufficiently early, its important connections with computer and software topics are not made sufficiently clear, and the material is not reinforced in computer courses. Many of these issues can be addressed by introducing discrete mathematics and logic as the first course for computer science and software engineering majors, and demonstrating the connections through the use of declarative based programming languages such as Prolog and Standard ML. The course Foundations of Computing is presented and experiences with this course are discussed.

1 Introduction

Computing Curriculum 2001 [3] is the first to require discrete mathematics in its core, thus recognizing the foundational nature of mathematics to the study of computer science. In a similar way, the emerging software engineering curriculum effort will include a focus on foundational mathematics. In addition, the Mathematics Association of America is interested in promoting and supporting the education of discrete mathematics in the U.S. for computer science, engineering, applied mathematics and other disciplines requiring discrete concepts [9] The goals of many of these efforts, and of other groups [2], [10], is to raise the level of mathematical rigor for these disciplines so as to ensure they are perceived as valid professional disciplines.

Over the past 20 years a large number of undergraduate level discrete mathematics textbooks for computer science have appeared on the market [5], [4]. Several of the more recent books include laboratory exercises, some of which use logic and functional programming languages to reinforce important mathematical concepts [6], [7]. Many of these books are introductory in nature.

There are several perspectives regarding the role functional and declarative programming play in computing education. One is a focus on the language as the object of study, that is, learning to program. Another is to use the language as a vehicle for reinforcing computer science and mathematical concepts - in the same way that Maple, Mathematica or Axiom are used in mathematics courses. Many introductory computer science courses take a middle road approach, trying to achieve both. However, in a discrete mathematics course, where the focus is on concepts rather than learning the language, the second view is more appropriate. This is the approach we have taken in our Foundations of Computing I course CS-151, the first course for computer science majors at Butler University.

This paper is organized as follows. The motivation for teaching discrete mathematics early and specific educational objectives are introduced. Next the topics covered in the Foundations of Computing I course are introduced, followed by the pedagogy underlying the use of declarative and functional languages for reinforcing these mathematics concepts. The use of theorem provers and ProLog for teaching logic is presented, followed by the use of Standard ML for teaching functions, recursive problem solving and mathematical induction. The conclusion wraps up the paper.

2 Motivation

Entering students are not capable of abstraction at the level necessary to compose good algorithmic solutions. This belief, coupled with very weak problem solving skills, a lack of basic mathematical maturity, poor oral and written communication skills, and finally, instructional programming languages which do not support abstraction principles well, places the entering student at a distinct disadvantage in a traditional introductory computer science course. Attend to the students' weaknesses first, then teach them programming.

The educational objectives of our entry level discrete mathematics course are: to impart fundamental discrete mathematics concepts, to develop mathematical maturity and mathematical thinking, to learn to use mathematics for modeling and problem solving, and to appreciate mathematics as a powerful tool for thinking, problem solving and abstraction. Students should become comfortable with mathematical symbols, notation and their manipulation. Recursive and inductive thinking, so important in computer science, are emphasized.

3 Foundations Curriculum

The Foundations of Computing I course covers the following topics: prepositional and predicate logic, proof techniques, sequences, sets, functions, binary relations, basic graph concepts, and recursive and inductive thinking. Problem solving and connections to applications are an integral part of the course.

Pedagogy There should be a synergistic relationship between the mathematical concepts and the educational tools used for conveying these concepts to students. Introductory mathematics is declarative in nature, making declarative oriented languages, including functional, better suited for reinforcing concepts. In addition, it is important to use the right tool/language at the right time. For example, theorem provers and logic programming languages are effective for reinforcing proposition and predicate logic concepts, a set based language (e.g., ISETL) for sets, and a functional language for reinforcing the concept of a function. Relations and graphs are easily represented in most logic and functional languages.

Mathematical recreation problems are used to introduce students to mathematical concepts and to illustrate the power of thinking mathematically. The same problem is viewed from a variety of mathematical and problem solving perspectives.

Teaching Logic. Basic prepositional and predicate logic are the declarative foundations upon which most computer science problem solving is based. For example, students must understand the truth or falsehood of statements like "The sky is blue." and "x = 6", and be able to formally express in logic the meaning of sentences such as "All cats have tails." This is analogous to constructing a math model of a logical English statement by translating that statement into a logic clause. The basic principles of proof require translating a set of statements and a conclusion into formal logic, and then performing a proof in this logical model. This seems to be difficult for most students because it requires a higher level of facility with mathematics and reasoning than many entering students feel comfortable with. To help overcome these weaknesses, the use of theorem provers and logic languages such as ProLog are used to more concretely reinforce these concepts and provide an interactive laboratory learning environment for students.

We have developed two simple resolution based theorem provers, one for prepositional logic and one for predicate logic. The input to these provers are a set of clauses (e.g., p, $p \vee q$, ~$p \vee q$, etc) and a conclusion (e.g., $p \wedge q$). The prover negates the conclusion and applies resolution steps until it either reaches a contradiction or halts when all possible resolutions have been tried - the conclusion can't be proven from the clauses. Each prover can be run in three modes: automatic with reports (explains each resolution step) automatic, and interactive mode in which students control the proof process (i.e., resolutions performed). For all, when a contradiction is reached, a resolution tree is displayed illustrating the entire proof.

In this laboratory students are required to translate prepositional logical problem statements into clauses and conclusions, and then use the theorem prover to see if a proof can be found. The subsequent laboratory does this for predicate logic using the predicate logic theorem prover. Students learn how to translate logical word problems and the limitations of theorem provers. The goal of these two theorem prover labs is to ensure students understand how to translate word problems to logic, understand the basics of proof by contradiction and logical resolution, and perhaps most important, backtracking in problem solving.

The next laboratory uses ProLog. Again, students translate logical word problems to prolog clauses and use Prologs query mechanism to initiate proofs in both prepositional logic and predicate logic. Many of these problems are identical to those they used in the previous theorem prover labs. However, additional problems such as the recursive definition of the ancestor predicate using the parent predicate are introduced. In addition, students define new predicates such as nephew, aunt, grandfather, etc. One of the goals of this laboratory is to ensure students understand the goal directed problem solving and backtracking approaches of Prolog, and to understand the connections between goal directed problem solving and resolution based problem solving.

Teaching Functions. Entering students have a very narrow views of functions. For example, $f(x) = x^2$ or $f(x) = ax + b$ with real valued domain and co-domain. We use this limited knowledge to build more general conceptual views of mathematical functions. To further reinforce these ideas and provide an interactive play ground for problem solving using functions the functionally expressive language Standard ML is used. It supports a natural expressive power that permits students to understand and create function definitions that are very concise and amenable to mathematical analysis. In addition, there is a close correspondence between the mathematical functional notation students learn (e.g., $f : real \to real$) and SMLs notation (e.g., `f : real -> real`), and SML enforces a set of typing constraints on students since it is a strongly typed language.

Once the basic definition of a function, their properties, and examples are introduced, students apply their understanding in a closed laboratory environment working in pairs. We do not teach students Standard ML, nor do we provide them with books or reference manuals. One beauty of SML is that students can learn the requisite features by experimenting with a small collection of representative examples. Accordingly, students use an interactive tutorial and perform a set of prescribed activities to introduce the basic notation, data types, operators, function composition and function definitions. This is discovery learning at its best!

Once they have mastered these basics, they progress to defining simple function definitions over the fundamental data types (int, real, Boolean, strings, char, etc.). Representative example problems include functions that return the length of the hypotenuse of a right triangle given the length of the other two sides, a function that returns the minimum of 4 integer argument values, finding the real roots of a quadratic equation (this function returns an ordered pair), and functions `even : int -> bool` and `odd : int -> bool` using function `mod` .

Next students are given simple recursive function definitions such as factorial, Fibonacci and Pascals Triangle, and asked to run and demonstrate understanding of each. They then compose their own simple recursive function definitions, first by completing missing parts of incomplete definitions and then developing their own complete recursive function definitions. This cumulates with a challenging activity such as composing recursive functions for Newton's method (two of the arguments are $real \to real$ functions, the function itself and its derivative) and for computing the number of ways change can be made for a specified amount only using US currency coins (penny, nickel, dime, quarter, and half dollar).

In the next laboratory session, students are introduced to lists and the primitive list processing functions first, rest, and append they will be using. Initially, we require students to use our primitives instead of the SML list processing functions *hd*, *tl* and operators `::` to constrain students to a simple, understandable (in English), set of primitives. Also, initially students use if-then-else for implementing case analysis rather than patterns.

Again students start with an interactive list processing tutorial and set of prescribed activities. They are introduced to exceptions (e.g., first of an empty list). Prior to this laboratory students have been learned weak mathematical induction, have been introduced to wishful thinking [1] as a problem solving tool, and have applied basic inductive thinking (i.e., identifying and specifying general patterns in problems). Now students are ready to compose their own list processing functions, most of which are recursive. First they complete partially defined functions such as `sum : real list --> real` and `reverse : 'a list --> 'a list`, then define functions for progressively more challenging list processing problems such as

```
select : int * 'a list --> 'a
firstN : int * 'a list --> 'a list
ordered : int list --> bool
merge : int list * int list --> int list
Exists: ('a -> boolean) * 'a list -> Boolean
```

etc. Students are required to prove the correctness of their recursive function definitions using mathematical induction (basically structural induction, but we don't use this term) and extensively test their definitions. This provides a basic introduction to verification and validation.

The final set of list processing problems emphasize problem decomposition via wishful thinking. That is, identifying natural sub problems (i.e., functions here) one wishes they had to solve the problem. Many of these 'sub' functions require recursive definitions. The relative ease of problem decomposition in a pure functional domain is one reason why functional languages are commonly used in introductory computer science courses.

This part starts with problems whose natural subfunctions have already been defined by the students. For example, function lastN is easily defined given functions reverse and firstN. The last set of problems require natural decomposition, and definition, validation and verification of the sub functions identified. One representative problem is finding the length of the longest non-decreasing subsequence of an integer list.

The final laboratory session deals with binary trees. Again, primitives left-subtree, right-subtree, `label : 'a BinaryTree --> 'a` and a binary tree constructor `bt` are provided and the pedagogical process is similar to the that described above for the first and second functional labs. Strong mathematic induction is used for the verification proofs. For binary trees the MacIntosh SML environment was enhanced to incorporate a visual graph editor and graph display features to facilitate the creation and display of binary trees for students.

Please note that the term "programming" was not used anywhere above. This is to reinforce the underlying mathematical notion of defining functions rather than the algorithmic ideas typically associated with programming.

4 Conclusions.

The author has been using the approach described for over 15 years to teach the mathematical foundations of computing to entering students to both the State University of New York at Stony Brook and Butler University, and has found it to be a very effective way to use declarative and functional approaches and languages for reinforcing important discrete mathematics concepts for entering students [8]. Students have a feeling of accomplishment when they have success in an interactive learning environment. This is much more effective than the dry traditional lecture, written homework assignment approach used in most discrete math courses.

References

1. Abelson, H., Sussman, G., and Sussman, J.: "Structure and Interpretation of Computer Programs, 2nd Edition," The MIT Press, 1996.
2. Baldwin, D, and Henderson, P.: The Math-thinking Discussion group web site http://www.math-in-cs.org/
3. Computing Curricula 2001 : December 15, 2001, http://www.computer.org/education/cc2001/
4. Epp, Susanna: Discrete Mathematics with Applications, Second Edition, PWS Publishing Company, Boston, Massachusetts, 1995.
5. Gersting, Judith: Mathematical Structures for Computer Science, third edition, W.H. Freeman, NY, 1996.
6. Hall, C. and O'Donnell, J.: Discrete Mathematics using a Computer, Springer Verlag, 2000.
7. Hein, J.: Discrete Structures, Logic, and Computability, Second Edition, Jones and Bartlett, 2002.
8. Henderson, P.: "'Foundations of CS 1' Stony Brook Alumni Survey", http://www.sinc.sunysb.edu/cse113/survey/
9. Kelemen, C., Tucker, A.B, Henderson, P.B, Bruce, K., Astrachan, O., Baldwin, D., Skrien, D., Van Loan, C.: Computer Science Report to the CUPM Curriculum Foundations Workshop in Physics and Computer Science, October 1999, http://www.cs.swarthmore.edu/ cfk/cupm2.pdf
10. Fisler, K.: "Formal Methods Education Resources" http://www.cs.indiana.edu/formal-methods-education/