Software is Discrete Mathematics

Rex L Page
University of Oklahoma
School of Computer Science
Norman OK 73019 USA
+1 405-325-5408
page@ou.edu

ABSTRACT

A three-year study collected information bearing on the question of whether studying mathematics improves programming skills. An analysis of the data revealed significant differences in the programming effectiveness of two populations of students: (1) those who studied discrete mathematics through examples focused on reasoning about software and (2) those who studied the same mathematical topics illustrated with more traditional examples. Functional programming played a central role in the study because it provides a straightforward framework for the presentation of concepts such as predicate logic and proof by induction. Such topics can be covered in depth, staying almost entirely within the context of reasoning about software. The intricate complexities in logic that mutable variables carry with them need not arise, early on, to confuse novices struggling to understand new ideas. In addition, because functional languages provide useful and compact ways to express mathematical concepts, and because the choice of notation in mathematics courses is often at the discretion of the instructor (in contrast to the notational restrictions often fiercely guarded by the faculty in programming courses), discrete mathematics courses, as they are found in most computer science programs, provide an easy opportunity to enhance the education of students by exposing them to functional programming concepts.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *correctness proofs, formal methods*. K.3.2. [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum*.

General Terms

Design, Languages, Verification.

Keywords

Functional programming, discrete mathematics, predicate logic, correctness proofs, formal methods, software engineering.

ICFP'03, August 25-29, 2003, Uppsala, Sweden.

1. INTRODUCTION

Educators have long argued about the value of mathematics in the study of computing. What the various participants in the argument mean by "mathematics," while not firmly tied down, generally boils down to the use of mathematical reasoning to prove theorems. In this sense, most computer science programs include very little real mathematics. Students do not learn much about constructing proofs when they study infinitesimal calculus for science and engineering, or when they study other mathematical subjects such as differential equations, linear algebra, statistical methods, and numerical analysis. Most computing programs require courses in most of these topics, and in taking these courses students learn important methods for solving certain kinds of problems, but they do not learn to construct mathematical proofs. They do not learn to think like mathematicians.

Should they? Should computing students learn to think like mathematicians? If so, why? To appreciate their cultural legacy? To support their study of computing? To make them more employable? All of these reasons have value, but the argument often focuses on how the study of mathematics affects the practice of software development. Observations of such effects bring substance to the argument. One of the goals of the Beseme Project (three syllables, all rhyming with "eh") is to provide observations of this kind.

2. PROJECT

The Beseme Project offers educational material for courses in discrete mathematics or more specialized topics, such as mathematical logic, along with some evidence of the effectiveness of the material. Most of the usual topics of an elementary discrete mathematics course are discussed in the Beseme materials, with a special emphasis on the concept of mathematical proof. The lecture notes elucidate several dozen proofs, and homework projects and examination questions provide opportunities for students to succeed in constructing their own proofs.

The intent is for students to gain experience with the thought processes of mathematicians. Because these thought processes apply in many problem areas, there is a wide range of interesting examples to choose from, many of which can illustrate, equally well, the basic ideas. Examples in the Beseme material come primarily from the realm of reasoning about properties of software artifacts.

It happens that almost all of these software artifacts are expressed in the form of inductive equations. That is, they are functional programs, and that makes it possible for functional programming concepts to play a central role in the study of discrete mathematics.

The project also provides data about the performance of students who have studied the Beseme materials and compares their performance to that of students who have studied the same topics from another point of view. The data lends credence to the idea that studying real mathematics benefits the practice of software development, and that the benefits are greater when students are allowed to see how the mathematical ideas they are studying apply to software artifacts.

The topics covered in discrete mathematics courses are fairly standard: logic, sets, relations, functions, proof methods including induction, combinatorics, discrete probability, graphs, trees, and recursion. Section 4 of this paper provides details about the coverage of these topics in the Beseme materials.

At the University of Oklahoma, where the Beseme Project has been conducted, discrete mathematics is a prerequisite for a course in data structures and algorithms. The data structures course addresses primarily implementation issues, and the course in discrete mathematics focuses on theoretical issues, with structures such as trees and graphs, and algorithmic concepts such as recursion figuring prominently in the coverage. One could view the discrete mathematics course as the theoretical framework for the data structures course.

Most of the work in the data structures course involves software development, and the grades students earn in this course depend heavily on their programming skills. For this reason, it seems reasonable to expect a correlation between effectiveness in software development and grades in the data structures course. One could use the data structures grades of students as estimates of their programming abilities.

The discrete mathematics course is taught in two sections. For the past six semesters, one of the sections has taken a traditional approach, as exemplified in the text of Rosen [12]. Here, for example, proof by induction is used to verify number theoretic results, such as the formula for the sum of the first n natural numbers.

Applications in the traditional course have a "pure math" flavor. When software is discussed at all, projects call for writing small programs, not reasoning about them. The study of logic and proof by induction comprises about a third of the material, and none of it involves reasoning about software artifacts. The other two-thirds concerns sets, combinatorics, probability, graphs, and trees, and the emphasis is on traditional aspects of these topics. Theorems about properties of trees would be more likely choices for examples than, say, theorems about applications of trees in software contexts, even though either choice might equally well illustrate the same mathematical concept.

The other section of discrete mathematics, the Beseme section, reverses the ratio. Logic and proof by induction comprise about two-thirds of the material, and the other topics about one-third. A variation in emphasis of this magnitude among the topics in a required course is not unusual at the university level. Different instructors have different interests and ideas about the relative importance of topics. Their courses reflect those interests.

In the Beseme course, almost every example in predicate logic and proof by induction involves reasoning about a software artifact, usually concerning a correctness-related property, but sometimes a property related to the software's use of resources. Beseme students study the traditional mathematical methods and concepts, but the examples they see as illustrations of these concepts come from a non-traditional collection.

Students see many examples of software, and all but a few of them are expressed as functional programs. The programming language chosen for the project is Haskell, but could just as well have been another language based on lambda calculus, such as ML, Scheme, or a host of others.

The fact that the software artifacts used in these examples are functional programs is not a primary point of emphasis. They are presented as inductive equations, and justified on the basis that any function meeting certain computational requirements would have to satisfy the equations.

For example, most students find it obvious that an operator (++) for concatenating two lists would have to satisfy the following equations.

$$(x : xs) ++ ys = x : (xs ++ ys)$$
$$[\,] ++ ys = ys$$

In these equations, colon (:) denotes insertion of a new element at the beginning of a list, square brackets delimit lists (used here to denote the empty list), x stands for an element of a list, and xs and ys stand for arbitrary lists.

The equations express certain properties of concatenation, and they are used as a starting point to confirm other properties of the operation. The fact that the equations provide a complete definition of concatenation is not the main point. The objective is to explore properties that the equations entail.

The focus is not directly on programming, but the students see dozens of inductive definitions of this sort, and the idea that they act as complete definitions of functions in useful software gradually emerges. Students gain some facility with elementary functional programming concepts, and their interest is piqued. Some of them later complain in the data structures course about the gratuitous difficulty of implementing, in a language like C++, the simple ideas they learned in discrete mathematics.

The software artifacts that become targets for the application of the principles of logic in the Beseme section include definitions of many reduction operations, such as computing the sum of a list of numbers, Boolean logic operations on lists, concatenation, list length, computing the maximum value in a list, and other reductions. They also include sorting, exponentiation in logarithmic time, vector addition, inner product, sequence reversal, building and searching AVL trees, and other important computations.

There are significant differences between the Beseme and the traditional sections of the discrete mathematics course in terms of concepts emphasized, and even more significant differences in terms of examples used to illustrate the concepts. Therefore, the subsequent data structures course is taken by two different populations of students: those who have studied discrete mathematics with an emphasis on reasoning about software (the Beseme group) and those who have seen a more traditional collection of examples in their study of discrete mathematics (the traditional group). One might be able to assess some of the effects of the two approaches by comparing the performance of these two groups of students in the data structures course. And, since the

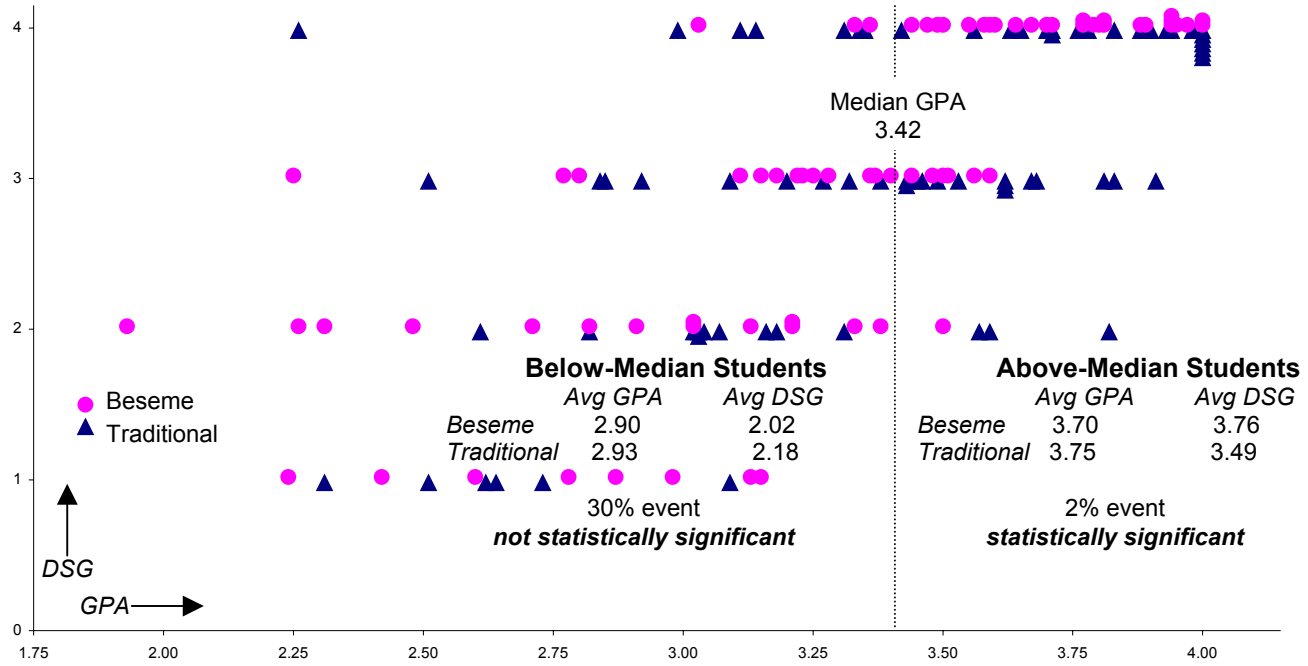

Figure 1. Grade Point Average vs. Data Structures Grade for Beseme and Traditional Students

data structures course has a heavy programming component, one might reasonably interpret affects on performance in the data structures course as effects on the programming skills of the students.

This paper takes that point of view. That is, it uses grades in the data structures course as estimates of the programming skills of students. Claims in the paper about programming skills are, more precisely, claims about grades in the data structures course.

3. STATISTICS

A common statistical method for comparing the average values of two random variables is to compute Student's *t* statistic from the data and to check how far out in the tails of its statistical distribution this measurement falls. Common standards call for rejecting the hypothesis that the two random variables have the same average value (the "null hypothesis") when the likelihood of the observed *t*-statistic (under the assumption that null hypothesis is true) is less than 5%.

The reliability of a statistical decision to reject the null hypothesis depends, in part, on the sizes of the populations of the two groups being compared. When both groups have populations exceeding 30, conclusions tend to be more consistently reliable than with smaller populations.

On the average, about eighty students per semester enroll in the data structures course at the University of Oklahoma. About two-thirds of these students have taken the discrete mathematics course during the previous semester, and another ten to fifteen percent have taken it two or more semesters prior to enrolling in data structures. A small percentage of students gain the right to enroll in data structures without having taken the discrete mathematics course.

These enrollment patterns mean that there is a lag of one or two semesters from the time the students complete the discrete mathematics course to the time they complete the data structures course. Data presently available consists of the records of 144 students who have passed both courses.

Figure 1 compares the performance of the Beseme and traditional groups. In this chart, student GPAs are plotted along the horizontal axis, and their grades in the data structures course are plotted along the vertical axis. GPAs are computed to three significant figures, but grades in data structures fall into only five categories (A, B, C, D, and F), so there is much less resolution along the vertical axis than the horizontal. Vertical bands in the chart contain data for students with similar GPAs. Not much can be perceived from this presentation of the data, but statistical computations yield some interesting results.

It happens that the 144 students are evenly divided between the Beseme group and the traditional group. This makes it possible to divide both groups into two subgroups and still maintain adequate populations for stable statistics within each of the resulting four subgroups.

The 72 students in the database from the traditional sections of discrete mathematics have an average GPA of 3.35 on a 4.00-point scale. These GPAs are computed for individual students upon completion of the data structures course by averaging all the grades they earned at the university, weighting each grade by the number of credits awarded for the course in which the grade was earned. A GPA of 4.00 indicates a grade of A in all classes taken by the student.

The 72 students in the Beseme group had an average GPA of 3.25. This was slightly lower than the 3.35 of the traditional group, but the difference is not statistically significant according to the *t*-statistic criterion discussed earlier.

Those are the average GPAs. With regard to the median, half of the 144 students in the database had a GPA exceeding 3.42.

The average grade earned in the data structures course by the 144 students in the database was 2.81 on a 4.00-point scale. The average data structures grade for the 72 students in the traditional group was 2.84, compared to 2.79 for the 72 Beseme students. This difference, like the difference in average GPAs for the two groups, is not statistically significant.

However, when the groups are divided into subgroups of more uniform ability (as estimated by GPA), statistical analysis yields more definitive conclusions.

Of the better half of the students, that is the 73 students with GPAs exceeding the median grade of 3.42, there were 39 from the traditional group and 34 from the Beseme group. The average GPA of the 39 above-median traditional students was 3.75. That figure for the Beseme students was 3.70, which is a little lower than the traditional group, but it is not a statistically significant difference. This means that the innate talent of the students in the two groups can be regarded as about the same.

The above-median students in the traditional group had an average grade in the data structures course of 3.49, compared with 3.76 for the above-median students in the Beseme group. According to the distribution of the *t*-statistic, there is less than a 2% likelihood that a difference this large would occur if the two groups represented samples from statistical populations with the same average grade in data structures.

Furthermore, since the students in this data set all had grade point averages exceeding 3.42, one would be very surprised if the average grade in data structures fell below 3.00. Since the probable range of the average is 3.00 to 4.00, the observed difference in the averages of the two groups, 0.27, seems significant in an intuitive sense.

A reasonable statistical interpretation of this result would be to reject the null hypothesis that the observations come from random variables with the same average. This means that one can, at a confidence level of 98%, accept the alternative that the better performance of the above-median students in the Beseme group is an effect arising from differences in the groups, not a random event. Using the data structures grade as an estimate of software development skills, as discussed earlier, one would interpret this to mean that Beseme students become better software developers than traditional students.

The average data structures grade for the below-median Beseme students was 2.02. Their average GPA was 2.90. This compares with an average data structures grade of 2.18 for the traditional students, who had an average GPA of 2.93. The distribution of grades in data structures for below-median students is more spread out than for above-median students. Because of the higher variance and the smaller difference between the average grades of the two groups, the difference is not significant statistically. That is, it is not a large enough difference to reject the null hypothesis at the 5% level. It is, in fact, a 30% event, which places it near the middle of the distribution, so it seems likely to be a random effect.

What factors explain the differences in the software development success of Beseme students compared with students of comparable talent in the traditional group? Three factors that might contribute to the observed differences include:

- material studied in discrete mathematics
- effectiveness of the discrete math instructor
- innate talent of individual students

With regard to innate talent, the average GPA of the Beseme students is slightly below that of the traditional students. The two groups are, apparently, more or less the same in terms of innate talent. So, innate talent is not a convincing explanation of the difference.

How about the instructor factor? As the instructor for the Beseme sections of discrete mathematics, I would like to claim instructor effectiveness as a reason for the better performance of the above-median Beseme students in the data structures course. However, students think otherwise. Near the end of each semester, students complete a questionnaire giving their assessment of various aspects of the quality of instruction. Their overall ratings of my teaching effectiveness in the discrete mathematics course averaged 2.17 on a 4.00-point scale (4.00 being the best rating). The ratings of the instructors for the traditional sections averaged 2.83.

Many instructors believe that student assessments of the performance of instructors are strongly influenced by the grades they award to students. This view holds that students will give better evaluations to instructors who award higher grades. The average grade awarded in discrete mathematics to the Beseme students was 2.83, compared with 2.96 for the traditional students. This difference falls near the middle of the distribution of the *t*-statistic. It is insignificant, statistically — probably a random event.

So, grading effects do not appear to explain the difference in instructor ratings. I have to accept the fact that students regard me as a less effective instructor than other members of the faculty who teach discrete mathematics. It's a bitter pill, but it means that the instructor effect fails to stand as a convincing explanation of the relative performance in the data structures course of students in the Beseme group compared with students in the traditional group.

That leaves course content in discrete mathematics as the primary suspect in the investigation of factors contributing to the better performance of Beseme students in the software development tasks required in the data structures course. Many explanations are possible, but this analysis suggests low credibility for the influence of two of them (innate student ability and instructor effectiveness). It seems reasonable to accept course content as an important factor contributing to the software development success of Beseme students.

4. MATERIAL

The Beseme course materials include over 350 animated slides. Many of the slides seem busy when viewed in isolation, at the end of the animation sequence, but the animation steps add information gradually, so that a step-by-step presentation can proceed at a comfortable pace. A typical slide, presented at a prudent pace through the animation sequence, with time for interactions with students along the way, takes five to ten minutes to discuss in a lecture.

The materials also include examinations (over 150 exam questions in all, with solutions), more than a hundred homework exercises and solutions, lesson plans, reading assignments, and software that checks proofs in propositional logic for correctness. (This software is an extension of tools provided with a textbook by Hall and O'Donnell [7]). All of the Beseme course materials are available to instructors through the Beseme website [9]. The website is password protected in the hope that instructors will feel comfortable using the materials in their courses. Selected materials may be viewed without a password and are provided to help instructors decide whether to request access to the full website.

In the Beseme course, all examples illustrating proof by induction are chosen from the realm of properties satisfied by software artifacts. Most of the analyzed properties confirm relationships between function inputs and results (that is, correctness issues). For example, early examples discuss properties of concatenation of lists, such as length-conservation and associativity. Later examples include verifying that a key present in an AVL tree will be found through binary search, with dozens of other examples in between.

Some examples discuss termination and performance properties. Those include, among other things, the logarithmic performance of the Russian peasant algorithm for exponentiation, the $n\ log(n)$ performance of merge-sort, and the logarithmic performance of AVL tree insertion.

In all, over two-dozen proofs by induction are carried out in lectures, and dozens more are required in homework and exams, all of which concern properties of software artifacts. About a third of these proofs use ordinary mathematical induction, $(P(0) \wedge \forall n.P(n) \rightarrow P(n+1)) \rightarrow \forall n.P(n)$, to verify that if a piece of software satisfies a few given equations, it must also have certain other desirable properties, which are stated in the theorem being proved.

Another third of the examples rely on strong induction, $(\forall n.(\forall m < n.P(m)) \rightarrow P(n)) \rightarrow \forall n.P(n)$. Induction on tree structures, and a form of induction on loops based on Floyd-Hoare logic are also illustrated in lectures, homework problems and exam questions.

Most of these examples involve reasoning based on a few equations that comprise an inductive definition of a function. The function might be selection of the maximum value in a sequence or merging two sequences, or some other useful computation. Proofs by induction are used to show that a function satisfying a few basic equations must also have other important properties.

The basic equations of an inductive definition are presented as reasonable properties that any function with the intended purpose would have to satisfy if it worked properly. The concatenation function mentioned in Section 2 provides an example. The basic equations that concatenation satisfies are repeated here with labels to facilitate referring to them in proofs.

$$(x : xs) \mathbin{++} ys = x : (xs \mathbin{++} ys) \qquad \{{++}\ :\}$$

$$[\,] \mathbin{++} ys = ys \qquad \{{++}\ [\,]\}$$

As noted before, the fact that these equations happen to form a complete definition of concatenation is not the main point. In fact, it is hardly mentioned. The goal is to show, through the application of logic, that the properties specified in these equations imply other properties of the concatenation operation, such as length conservation, associativity, etc.

One approach to verifying properties implied by these equations relies on induction over list structures. However, ordinary induction over the natural numbers can also be used, at least when the lists involved have finite length. Ordinary mathematical induction happens to be the topic of study at the time these equations appear in the Beseme course, so that is the method used to prove properties of concatenation. The following sketch of a proof along these lines illustrates the approach taken in the Beseme materials.

Prior to discussing the associativity of concatenation, a length conservation property is proved:

$$length(xs \mathbin{++} ys) = length\ xs + length\ ys$$

Associativity is then proved through induction on the length of xs in the equation expressing associativity:

$$xs \mathbin{++} (ys \mathbin{++} zs) = (xs \mathbin{++} ys) \mathbin{++} zs$$

The inductive case in the proof of associativity shown in Figure 2 makes use of equational reasoning. Reasons justifying each step in the sequence of equations are noted on the right-hand side of the figure. The argument applies in the case when the leftmost operand is nonempty. The proof for an empty leftmost operand is shorter. It consists of two applications of the {++ [ ]} equation, with no need to cite the induction hypothesis. In the Beseme materials, most proofs of properties derived from inductive definitions make use of syntax-driven substitution in equations as illustrated in this example.

$$\begin{aligned}
&(x: xs) \mathbin{++} (ys \mathbin{++} zs) \\
&= x: (xs \mathbin{++} (ys \mathbin{++} zs)) && \{{++}\ :\} \\
&= x: ((xs \mathbin{++} ys) \mathbin{++} zs) && \textit{induction hypothesis} \\
&= (x: (xs \mathbin{++} ys)) \mathbin{++} zs && \{{++}\ :\} \\
&= ((x: xs) \mathbin{++} ys) \mathbin{++} zs && \{{++}\ :\}
\end{aligned}$$

Figure 2. Associativity of Concatenation, Inductive Case

Most students are familiar with reasoning based on equations from their experience in high-school algebra. This much comes to them easily. In fact, one of the few examples in the course that does not involve a piece of software is a proof that the product of two negative numbers is a positive number.

This proof serves to introduce the idea of equational reasoning, and it has a profound effect. Most students have no idea why negative-times-negative is positive, and they are delighted to learn that this rule is only a few equations away from more basic rules of arithmetic such as the associative and distributive laws.

They see from this example how a formal argument based on equations can be constructed, and they learn to apply this idea with new equations and strange operations like list-insertion and concatenation, just as they formerly applied the ideas with more familiar operations, such as the addition and multiplication of numbers.

What does not come so easily to most students is the idea of proof by induction. This takes a lot of thinking and experience, and that is why the concept is presented in four guises (ordinary, strong, tree, and loop) with a couple dozen examples in lectures, and many more than that in homework and exams. Gradually, students

begin to grasp both induction and the idea of formal proof based on syntactic substitution in equations.

Equational reasoning and proof by induction comprise about thirty-five percent of the Beseme material, but it is not the only basis for reasoning that the students see. About thirty percent of the material focuses on natural deduction with Gentzen-style inference rules in propositional and predicate logic.

Software involving mutable variables provides another way for the students to gain experience in the process of conjuring up proofs, in this case relying on a form of induction for looping software based on Floyd-Hoare logic. A little less than ten percent of the material is devoted to this topic.

The different forms of induction reinforce each other. Students end up getting a lot of experience with inductive reasoning, and it helps them understand other concepts related to induction, such as recursion, numeric recurrence equations, and loop invariants.

The basic operations of set theory (union, intersection, difference, power sets, and so on) are discussed along with proofs about properties of sets. Functions and relations are discussed in set theoretic terms, along with special attributes (injective, surjective, reflexive, symmetric, transitive, etc.) and analytic tools (image, inverse image, etc.). These ideas comprise about fifteen percent of the lecture material, and they emerge repeatedly in other parts of the course, usually in a software context.

The idea of comparing asymptotic rates of growth ($f = O(g)$) helps describe algorithmic complexity, and several examples in this area serve to illustrate the estimation of algorithmic performance and solving recurrence equations. This coverage comprises a little more than ten percent of the material.

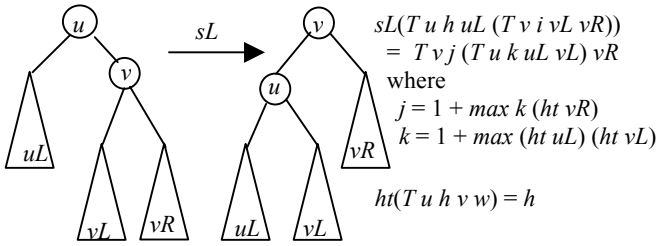

Figure 3. Simple Left Rotation

Of the traditional list of topics in discrete mathematics courses (logic, sets, relations, functions, proof methods including induction, combinatorics, discrete probability, graphs, trees, and recursion), only probability is entirely missing from the Beseme materials, although combinatorics gets short shrift. This is only fifteen-week, three-credit course, after all.

Because most of the software artifacts presented to illustrate the use of logic in the Beseme course are expressed in the form of inductive equations, they tend to be short. Most of them are two or three lines long. They exceed six lines in only one case. That case is AVL tree insertion. The equations for AVL tree insertion cover eleven lines. They are short lines. They fit on a standard slide to be displayed with an ordinary video projector. They are readable from the back of the room, so they use large print, which means that the lines have to be short.

AVL tree insertion is more complicated than concatenation. However, there is a straightforward, entirely formal (that is, mechanical) translation of the usual rotation diagrams into inductive equations. The simple case for left rotation is illustrated in Figure 3. The equations in this figure use a tree constructor *T*, which builds a tree from its root key, height, and left and right subtrees. (*T* is too cryptic. The Beseme presentation uses more descriptive names. Shorter names are used here to make the equations fit in one column of the required document style.) The function *ht* in these equations extracts the height component from a tree structure.

Since the more complex rotations amount to a composition of simple rotations, they are equally easy to derive from diagrams. Given this information, and an understanding of how to use left and right rotation (*l* and *r*) to ensure AVL balancing, most students can derive the five-case equation for AVL tree insertion (^:) shown in Figure 4.

```
z ^: (T u h uL uR) =
  if       z < u  &&  (ht vL) <= (ht uR)+1 then  T u i vL uR
  else if  z < u  &&  (ht vL) > (ht uR)+1  then  r(T u i vL uR)
  else if  z > u  &&  (ht vR) <= (ht uL)+1 then  T u j uL vR
  else if  z > u  &&  (ht vR) > (ht uL)+1  then  l(T u j uL vR)
  else  error "key already present"
  where
     vL  =  z ^: uL
     i  =  1 + max (ht vL) (ht uR)
     vR  =  z ^: uR
     j  =  1 + max (ht uL) (ht vR)
```

Figure 4. AVL Tree Insertion

Induction on AVL trees (yet another kind of induction) is used to verify that a tree that is balanced before insertion remains balanced after insertion, that the time required for insertion is proportional to the height of the tree, and that the number of nodes in a balanced tree is exponential, compared with its height. This leads to the conclusion that AVL insertion requires computation time proportional to the logarithm of the number of keys in the tree. Recurrence equations needed for these arguments match the structure of the inductive equations for insertion.

Inductive equations are common in mathematics, and the fact that these happen to be presented within the syntax of a programming language might be viewed as more-or-less incidental. In practice, however, it is more than incidental because students can run programs directly from the equations. They also begin to see how significant pieces of software can be constructed from inductive equations. That is, they acquire some experience with functional programming. It occurs as if by osmosis, but statistical analysis of the Beseme data suggests that it has a significant, positive effect, at least on the upper half of the students.

5. RELATED AND FUTURE WORK

The Beseme Project is one of many efforts emphasizing the use of logic in software development. Dean and Hinchey [3] edited a collection of papers presenting projects with goals related to those of the Beseme Project. There are also several textbooks that, to

varying extents, discuss applications of logic to reasoning about software ([2], [4], [5], [6], [7], [8]).

The TeachLogic Project [1] based at Rice University is developing materials similar to those of the Beseme Project, using Scheme as the notation for inductive definitions instead of Haskell, which is the notation used in the Beseme Project. The TeachLogic Project aims to provide two- to four-week modules that can be used in mainstream computing courses, such as programming languages, databases, artificial intelligence, and discrete mathematics, while the Beseme project focuses on an extensive treatment in one of these areas. Both projects share the goal of presenting logic to students in the context of software development.

The Beseme Project has, so far, assessed results only in terms of grades in the data structures course. As the students progress further through the curriculum, it will be interesting to see if a larger pattern of effects can be observed by looking at grades in other courses.

Part of the motivation of the Beseme Project is to provide materials that can help instructors introduce functional programming to students in ways that can benefit them. The expectation would be that students with this background might be inclined to make use of it later, and that might lead to substantial improvements in software quality.

Barriers to the use of functional programming for developing software in industry are not as high as many suppose. Many software development groups have license to choose their implementation tools [10]. Support for functional programming in terms of libraries and communication with other software is manageable in a practical way for many projects.

So, why don't software developers choose functional languages more often for their projects? There are many answers to this question [13], one of which is that only a tiny percentage of software developers receive any serious exposure to functional programming in their education. What they are unfamiliar with, they are unlikely to choose.

The Beseme Project provides at least a little material to fill some gaps. It is part of a larger plan that envisions a core curriculum for a programming-oriented baccalaureate that might be dubbed "hard-core software engineering."

Dictionaries and engineering organizations define "engineering" as the use of mathematics and science to design useful artifacts. A straightforward extension of this definition would imply that software engineering is the use of mathematics and science to design software. However, that is neither the general understanding of the term "software engineering," nor does it well describe the usual practice of software engineering.

The hard-core software engineering curriculum will appeal to those who think software engineering would be better practiced if it, like other engineering disciplines, consisted primarily in the application of mathematical principles to the design of useful artifacts, which in this case would be software rather than bridges or radios or other useful things.

A software engineering program at McMaster University [11] has similar goals to those of the program envisioned as an extension of the Beseme Project. In the Beseme version of this approach, six courses occurring in three, closely coupled pairs, would form the core of the curriculum.

Within each pair, there would be a mathematics course and a programming course. Students would enroll in the two courses during the same term, and the material of each of the courses would complement that of the other. All of the mathematics would be illustrated in terms of the software concepts being discussed in the programming course, and all of the ideas in the programming course would be explained in terms of the principles being discussed in the mathematics course.

The first pair would be a functional programming course, together with a discrete mathematics course along the lines of the Beseme materials. The second pair would cover aspects of graph theory and abstract algebra in the mathematics component, and would cover data structures in the programming component, using, initially, functional programming, but gradually introducing conventional programming techniques.

The third pair of courses would cover a formal model for concurrency, such as Milner's pi calculus, in the mathematics component, and would cover the design and architecture of operating systems in the programming component. The programs in the operating systems course would be written in a conventional programming language.

These six courses would form a basis for the rest of the computing curriculum, which could consist of a selection of more-or-less standard, upper division, computer science courses. The core courses would place students on a firm footing for their study of the software enterprise, including both the theoretical and the practical aspects of computing. The program is envisioned as a software-oriented one, and the degree title "software engineering" seems appropriate, even if the core material does not have the generally accepted shape of software engineering as commonly understood and practiced.

The first integrated pair, discrete mathematics and introductory functional programming, is scheduled to be offered at the University of Oklahoma in the coming academic year. Progress on the other two pairs awaits the time and energy required for their development, and depends, in part, on success in the delivery of the first pair.

6. CONCLUSION

The Beseme Project has delivered a body of materials for courses in discrete mathematics. The materials include lecture notes, lesson plans, reading assignments, homework (and solutions), exam questions (and solutions), and software tools that have been tested across six semesters of use in the classroom. The material focuses on reasoning about software, and most of that software is expressed using the functional programming paradigm. The approach provides an example of exposing students to functional programming concepts at a point in the curriculum that provides an excellent, but often overlooked, target of opportunity.

Functional programming is a crucial element in the process. Without it, things get too complicated, too fast, and students tend to miss the point. Functional programming provides a context in which student interest remains high as the course progresses and in which most students can succeed.

Beseme Project results support the conjecture that experience in the direct application of logic to reasoning about software leads to

increased effectiveness in the practice of software development. A statistical analysis of data on student performance in software development projects indicates that students whose educational background includes experience in applying mathematical logic to the problem of reasoning about properties of software get better grades in a subsequent, programming-intensive course than students who studied mathematical logic in more traditional contexts.

The data supports this conclusion at a 98% confidence level for students in the upper half of the distribution of academic talent. No statistically significant effect was observed for below-median students. This outcome was observed through data collected from a population of 144 students in two courses, one in discrete mathematics and the other in data structures. Half of the students studied discrete mathematics with an emphasis on logic applied to reasoning about software, and the other half took a traditional discrete mathematics course. All of the students went on to study implementations of data structures in a course with a heavy software development component.

Performance in the software development course was used to estimate the software development abilities of the students. Factors such as innate academic ability and the quality of instruction in the courses on which the study was based appear to have played no significant role in the outcome.

A single study of the impact of particular choices of educational materials requires reinforcement from other quarters if it is to have an influence on computing education. If the results of this study make it easier for other educators to introduce reasoning about software in their own courses, then the Beseme Project may contribute to the goal of better software engineering through mathematics education.

7. ACKNOWLEDGMENTS

This material is based on work supported by the National Science Foundation under Grant No. EIA 0082849. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

I am grateful to John Canning for sharing in the design of the Beseme Project and the writing of the proposal that led to funding, even though he was not able to join the project when it got underway. I am also grateful to Jeffrey Sharp, who wrote the software for the database of statistics used in this work, to Justin Beitelspacher, who maintained the database software and developed the Beseme website, and to Shauna Singleton for entering the data and providing it to the research staff, stripped of all student identification information.

Two students deserve credit for extending the proof-checking software provided with the Hall-O'Donnell textbook: Pierre Lemaire added the ability to cite proven theorems in proofs by natural deduction, and Jonathan Cast added the ability to check equational proofs, in addition to the support for natural deduction provided in the original software. Finally, I want to thank the referees, all of whom made suggestions that improved the quality of this paper.

8. REFERENCES

[1] Barland, I., Felleisen, M. Kolaitis, P., and Vardi, M. TeachLogic Project, http://www.cs.rice.edu/~tlogic

[2] Broda, K. Eisenbach, S. Khoshnevisan, H., and Vickers, S. Reasoned Programming, Prentice Hall, 1994.

[3] Dean, C., and Hinchey, M. (eds.) Teaching and Learning Formal Methods, Academic Press, 1996.

[4] Grassman, W., and Tremblay, J-P. Logic and Discrete Mathematics: A Computer Science Perspective, Prentice Hall, 1996.

[5] Gries, D., and Schneider, F. A Logical Approach to Discrete Math, Springer-Verlag, 1993.

[6] Hein, J. Discrete Structures, Logic, and Computability, 2nd Edition, Jones and Bartlett, 2003.

[7] Hall, C., and O'Donnell, J. Discrete Mathematics Using a Computer, Springer, 2000.

[8] Huth, M., and Ryan, M. Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, 2000.

[9] Page, R. Beseme Project, http://www.cs.ou.edu/~beseme

[10] Page, R. Functional programming ... and where you can put it, ACM SIGPLAN Notices 36, 9 (September 2001) 19-24.

[11] Parnas, D. A software engineering program of lasting value, in Proceedings of AMAST 2000 (Iowa City IA, May 2000), Lecture Notes in Computer Science 1816, Springer, 2000.

[12] Rosen, K. Discrete Mathematics and Its Applications, McGraw-Hill, 1999.

[13] Wadler, P. Why no one uses functional languages, ACM SIGPLAN Notices 33, 8 (August 1998) 23-27.