Teaching Discrete Structures with SML

Christelle Scharff[1] and Andrew Wildenberg[2]

[1] Pace University, Department of Computer Science, New York, NY, USA, cscharff@pace.edu
[2] State University of New York at Stony Brook, Department of Computer Science, Stony Brook, NY, USA, awilden@cs.sunysb.edu

Abstract. We describe the benefits of using SML to teach the Discrete Structures course described in the ACM Computer Science curriculum. When students in a Discrete Structures course have no prior programming experience, SML can be used to familiarize students with recursion and to illustrate various topics introduced in the course without teaching programming per se. At the same time, it reinforces the links between mathematics and CS, making Discrete Structures more relevant. SML also makes it easy to deliver and automatically evaluate online or web-based homeworks, such as with the WeBWorK system.

1 Introduction

According to the ACM standards for computer science education, Discrete Structures is a required course, typically taught in one or two semesters [5, 12, 13]. Two important subtopics of Discrete Structures are functions and recursion. Both are fundamental concepts in Computer Science that appear throughout the curriculum.

As a new approach for problem solving, recursion is difficult for students – students typically do not see recursion as a natural way to solve problems and they don't see how it can work [9, 10, 14]. Visualization in assisting the use of recursion through software and web-based tools can help students learn and understand recursion [7, 16]. Previous interactive systems for teaching recursion include [1, 3, 7].

In this paper we describe how we use SML (Standard Meta Language) [15] to teach functions and recursion in the Discrete Structures course at the State University of New York at Stony Brook. Because of an unusual ordering of courses at Stony Brook, Discrete Structures is the first course taken by students who want to enter the Computer Science or Information Science majors – students typically have no programming experience.

To address these issues, we teach students to use a minimal subset of SML. Because SML uses mathematical syntax and notation, students can start writing functions very quickly, and its syntax and semantics reinforce many of the mathematical concepts covered in Discrete Structures. Since SML is an interactive functional programming language [6, 8], students can interact with functions without writing full programs, avoiding the need to explain the edit/compile/execute cycle. SML's type safety and completeness rules help catch many errors during the function definition stage, simplifying debugging.

Most importantly, writing SML functions helps convince students that the mathematics in Discrete Structures is actually relevant to their careers as computer scientists. Many undergraduate computing curricula teach mathematics in isolation from computing, leaving students understanding little about how and why mathematics applies to computing problems [2, 17]. Our course addresses this issue by having students define functions in SML in order to solve mathematical problems. The emphasis is not on presenting SML as a programming language but as rather as a language to define and experiment with functions and recursion.

Problems on recursion and SML have been written in the WeBWorK web-based homework system [11] for online experimentation, grading and support. WeBWorK features randomized problems (each student gets a different but equivalent problem) and instantaneous grading feedback. Student feedback on WeBWorK has been nearly universally positive – students enjoy using the system and feel it teaches them well.

Over the past three years, this integration of SML into Discrete Structures has been used by the authors at SUNY Stony Brook to teach more than 2000 computer science, information science and computer engineering students. Others at Stony Brook have taught with SML and a similar curriculum since 1988.

2 Discrete Structures for Non-Programmers

The sequence of introductory courses of Computer Science at State University of New York at Stony Brook is composed of a first Discrete Structures course (the one presented in this paper), a programming course in JAVA, a second Discrete Structures course emphasizing predicate logic and a data structure courses based on JAVA. The two last courses can be switched. Because Discrete Structures I is taught before any programming course, roughly two-thirds of the students have no programming experience.

Discrete Structures I is organized around the following sequence of topics: propositional logic, number theory, set theory and functions, recursion, functional programming (FP) and SML, induction, correctness of SML programs, automata, trees and graphs. Early in the course, students are taught the basics of formal systems and proofs in propositional logic – these forms the basis for more complicated proofs in number theory, set theory and induction covered later in the course.

Recursion, as a new approach for problem solving, is emphasized and its use is illustrated through FP. We show how the FP paradigm is related to the concept of functions that we introduced earlier in our course, how mathematics is useful in Computer Science and how a FP language is convenient for solving problems related to Discrete Structures.

Students are taught to use SML as a tool to define functions and experiment with them rather than as a language to write complete programs. The built-in *list* type is used to illustrate recursive functions on non-numerical domains on topics including the Towers of Hanoi and Merge Sort – topics that are not typical list-operations. We also teach correctness proofs of simple SML programs using induction.

3 SML for mathematics

There are many characteristics of SML that lend itself to teaching SML to non-programmers. One of the most important is that the syntax of SML is closely related to standard mathematical typesetting conventions. One distinct advantage to teaching discrete mathematics with SML is the simplicity of SML's syntax. Functions are defined in a natural way with a minimum of keywords and new symbols.

Table 1 shows two simple functions defined in SML and typeset as in a mathematics textbook. While some differences exist between the two, basic SML syntax is simple enough that it requires no explanation to students already familiar with mathematical functions. In fact, the lecture notes actually start using the notation (without saying so) two lectures *before* the first lecture on SML. It is only later that students are shown that they've actually been reading SML.

Table 1. SML and typesetting look alike *The types and definitions of two sample functions are presented using normal mathematical typesetting and in the SML programming language. Because of the similarities between the two, students typically can read, write and use SML functions with very little background.*

Definition	$f(x,y) = x^2 + y + 1.5$ `fun f(x,y) = x*x+y+1.5;`
Type	$f : (\mathbf{R} \times \mathbf{R}) \longrightarrow \mathbf{R}$ `val f = fn : real*real -> real`
Application	$f(2.1, 3.2)$ `f(2.1,3.2);`

Definition	$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$
	 `fun M(n)=if (n>100) then n-10`
	 `else M(M(n+11));`

Type	$M : \mathbf{Z} \longrightarrow \mathbf{Z}$ `val M = fn : int -> int`
Application	$M(88)$ `M(88);`

Compared to most languages for teaching programming to beginning students, SML is much closer to the standard mathematical syntax and semantics. Table 2 compares the implementation of a simple function in different functional languages typically used to teach introductory programming. SML is nearly identical to the mathematical typesetting in the function's definition, type and application. Simple functions defined in procedural languages such as C, C++, Java and Ada also tend to be more complex than in SML – and code in these languages is less interactive because of their inherent edit/compile/execute cycle.

To minimize the amount of programming that needs to be taught to students, only a small subset of SML is presented to students. Specifically, typedefs, exceptions, let statements, packages and functors are not taught, while list operations, tuples, pattern matching function definitions and type inference are. Although this limits what students can do with the language, it is more than enough for pedagogical purposes and it means that the course can spend its time teaching the relevant mathematics instead of programming.

There are other benefits to this subset of SML. Students are forced to use recursion to solve all but the simplest problems, unlike most programming languages where they can fall back on iterative constructs. The resulting functions are usually total functions in the mathematical sense: they are free of side-effects, and partial functions are discouraged. For example, `fun f(3)=4;` generates warnings or errors in SML because they are not defined for the entire domain. Many CS students are repeatedly shown how a (programming language) function can have arbitrary side effects (for example the `rand()` function that returns random numbers – a different value every time it is called). By teaching functions that are free of side effects just like mathematical functions, students have one less case of "cognitive dissonance" to overcome.

SML's emphasis on type reinforces the mathematical concepts of domain and codomain. As each function is defined, SML reports the name and the type as `fn : A -> B` where `A` is the function's domain and `B` is the function's codomain (for example, see Table 1). Parameters in SML are call by value, which matches the standard mathematical semantics. Students are told when they write functions to try and determine the domain and codomain of the function before they begin. They are taught to use basic parameter type declarations to help SML verify their decisions.

Type and type inference is one of the first emphases in teaching SML. Students are taught simple types, tuples and lists. Tuples and lists are compared with the definition of mathematical sets from earlier in the course. Oftentimes the homework problems include defining standard set operations (union, intersection, Cartesian product) using SML lists.

The FP paradigm and the basic mode of computation in ML, as in other functional languages, is the use of the definition and application of functions

Table 2. A comparison of function implementations in several FP languages *The function f from table 1 is implemented in several functional languages commonly used to teach programming. SML's definition is the most similar to the original typesetting.*

Original	$f(x, y) = x^2 + y + 1.5$
SML	`fun f(x,y) = x*x+y+1.5;`

Scheme	`(define f`
	`   (lambda (x y)`
	`      (+ (* x x) y 1.5)`
	`)  )`

Logo	`to f :x :y`
	`output :x*:x+:y+1.5`
	`end`

Mathematica	`f[x_,y_]:=x*x+y+1.5`
Matlab	`function o = f(x,y)` `o=x*x+y+1.5;`

(explicit and recursive). Students are shown how this makes the system interactive and how this enables experimentation.

Function type is discussed and recursive function definition is illustrated through numerical (factorial, Fibonacci) and non-numerical examples (list operations, Towers of Hanoi, Merge Sort). A lot of time is spent discussing how an original problem can be divided up and how that division is dependent on how the solutions can be "glued" back together. Higher-order functions, polymorphism and mutual recursion are shown to be useful examples of this "glue". The notion of pattern matching is presented as a new way to look at algorithms.

4 SML and Online Homework

As part of their homework, students do WeBWorK [11] problems that encourage experimentation on the fundamental concepts of SML by students. Developed by the University of Rochester, WeBWorK is a web-based method for delivering and evaluating homework problems. It provides user authentication, instant grading, individualized problem sets, the evaluation of free-form symbolic answers (i.e. formulas) and the distribution of solutions. WeBWorK is used to teach a variety of math and science classes at major universities across the country.

The WeBWorK problems on SML contain problems on fundamental problems on SML as typing and evaluation of recursive functions as well as SML terminology. Mathematical concepts of domain, codomain and composition of functions are covered. The simple syntax and semantics of SML means that the problems are concise, easy to write and easy to evaluate. Figure 1 presents some WeBWorK problems on SML.

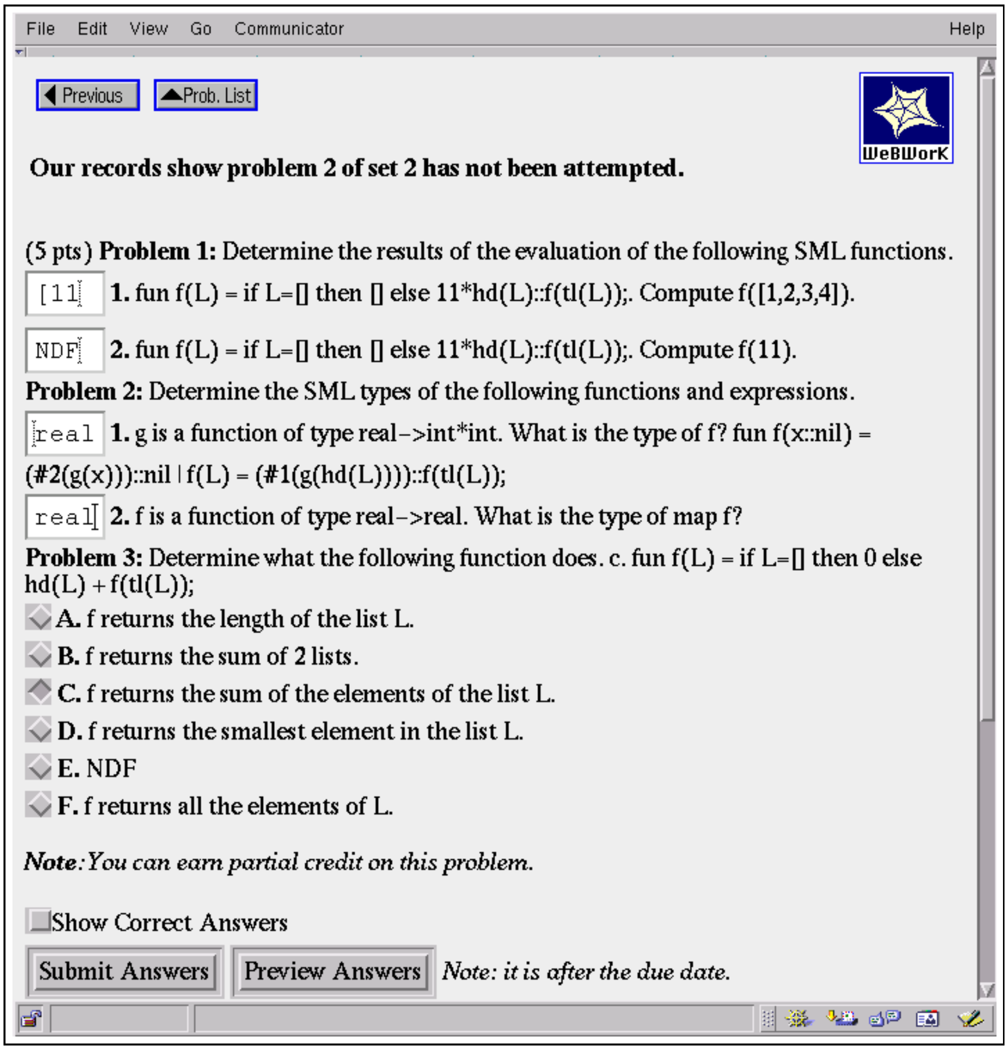

Fig. 1. A screen shot of WeBWorK in use

5 Pedagogical goals

SML is used primarily to encourage mathematical thinking and improve the understanding of recursion. Our general approach is to teach first recursion free of any language. Later students can look at problems in a programming language to further their understanding.

Of course SML is a good language for topics beyond teaching recursion. Algorithms are expressed very naturally using rules and patterns. Students are introduced to design, prototyping and software quality in terms of modularity and reliability. They are encouraged to exhaustively test their functions and this task is very direct in SML. Because SML avoids side-effects (in the subset we teach) students can use induction to prove formal correctness of SML functions without a lot of background – this is typically taught in a single lecture.

6 User feedback and analysis

To assess the effectiveness of the SML/Discrete Structures curriculum, we surveyed students in the class, as and tried to collect anecdotal evidence.

Before the class only 34.5% of the students had used or heard about recursion but after the class 64% of the students were comfortable with it. 64% roughly corresponds to the percentage of students who receive the C or higher that they need to advance in the CS curriculum. This is a great improvement considering the difficulties students always encounter with recursion.

Anecdotal evidence also supports the effectiveness. There seems to be a strong link between students who attempt the SML programming assignments and those who do well on recursion problems on the exam – when the students actually do the work they seem to learn recursion well. Professors teaching later courses also believe in the methodology. In one case, when the department considered making Programming I and Discrete Structures corequisites, the Programming I instructor threatened to never teach it again because he was worried about the dropoff in the quality of students, specifically citing their understanding of recursion.

However, SML is not very popular with students. 70.3% of students said they did not like SML and 54% admitted to bringing a negative preconception of SML into the class. They feel SML is more linked with mathematics than computer science (54%) and feel SML is difficult to learn. When asked to explain their difficulties, the top two reasons listed were recursion (75%) and problem solving (40%), indicating that the real difficulties may not actually related to SML but with recursion itself. This is consistent with student comments in office hours. One of the most common comments was "I could do this so easily in C/C++/Java" but when asked for solutions to the problems in those languages, they invariably produced iterative solutions instead of recursive ones.

On the other hand, WeBWorK was universally popular. Only 22% of students said they preferred paper-based homework to web-based homework. Only 8% of the class said they felt it was easy to cheat on the homework.

7 Future directions

When students were asked what could improve their interactions with SML, many said they wanted a more friendly user interface (80.5%). They also said they had difficulty understanding error messages and debugging programs. To address these concerns, we hope to develop a front end, similar to the AdaGIDE system[4], to provide a friendlier interface for novice users and help students interpret error messages. This would likely be built upon the current web-based SML program submission system and its semi-automatic grading system.

We also hope to expand the use of SML in other parts of the computer science curriculum. As a first step, SML is being used at Pace University in a course called 'Languages and Implementations' to introduce students to a new programming paradigm while reinforcing their understanding of recursion, patterns, testing, prototyping and correctness. In the same course WeBWorK has been used to deliver more advanced SML problems.

8 Acknowledgments

The authors are greatly indebted to Peter Henderson who first introduced SML into the State University of New York at Stony Brook Discrete Structures class in 1988, and Leo Bachmair and Steve Skiena who did a significant rewrite of the curricula in 1997. The authors would like to acknowledge Dean Susan Merritt, Dean Dennis Anderson and the Network and IT support people for their support in the development of a course called 'Languages and Implementations' around SML and WeBWorK at Pace University.

References

1. H. Abelson and A. diSessa. *Turtle Geometry – The Computer as a Medium for Exploring Mathematics*. MIT Press, 1981.
2. D. Baldwin and P. Henderson. The importance of mathematics to the software practitioner. *IEEE Software*, March/April 2002.
3. G. Brownell. Logo as a language to teach non-majors the essentials of programming. *Journal of Information Science Education*, 1(1), September 1988.
4. M. C. Carlisle and A. Chamillard. Adagide: A friendly introductory programming environment for a freshman computer science course. *Ada Letters*, 18(2):42–52, March 1998.
5. Computing curricula 2001. http://www.acm.org/sigcse/cc2001/ , December 2001.
6. G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

7. W. Dann, S. Cooper, and R. Pausch. Using visualization to teach novices recursion. In *Proceedings of 6th Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 109–113, July 2001.
8. A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
9. H. Kahney. What do novice programmers know about recursion? In *Proceedings of the CHI 83 Conference on Human Factors in Computer Science*, pages 235–239, 1983.
10. H. Kahney and M. Eisenstadt. Programmers' mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. In *Proceedings of the 4th Annual Conference of the Cognitive Science Society*, pages 143–145, 1982.
11. A. Pizer and M. Gage. The webwork system. http://webwork.math.rochester.edu.
12. E. Roberts, C. Cover, C. Chang, G. Engel, A. M. Gettrick, and U. Wolz. Computing curricula 2001: How will it work for you? In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 433–434, 2001.
13. E. Roberts, R. Leblanc, R. Shackelford, P. Denning, P. Srimani, and J. Croos. Curriculum 2001: Interim report from the ACM/IEEE-CS task force. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, volume 199, pages 343–344, New Orleans, Louisiana, 2001.
14. R. Sooriamurthi. Problems in comprehending recursion and suggested solutions. In *Proceedings of 6th Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, July 2001.
15. J. Ullman. *Elements of ML Programming*. Prentice Hall, 1998.
16. D. Wilcocks and I. Sanders. Animating recursion as an aid to instruction. *Computer Education*, 23(3):221–226, 1994.
17. J. Wilkinson, R. Matthew, and H. Earnshaw. Engineers need mathematics but can we make it interesting? In *Proceedings of the International Conference on Engineering Education*, pages 15–20, 2001.