Selling Haskell for CS1

Rex Page *

University of Oklahoma

Abstract

A prominent role for the functional paradigm is rare in computing programs. Introducing the idea is difficult, partly because so few programs use functional methods that qualified instructional staff, at both the faculty and graduate assistant level, are scarce. This paper outlines some ways to facilitate moving towards the use of the declarative paradigms early in the curriculum, presents arguments in favor of doing so, discusses frequently heard arguments against the idea, and describes some experiences with CS1 courses using Haskell.

1 Introduction

Computing educators have arrived at no consensus about either the content or the primary emphasis of the first computing course, often referred to as CS1 [13]. Some think it should focus on teaching novices to write software [9], and others would like it to be a broad introduction to the central concepts of computing, with programming as one of the activities [14]. Either way, the course involves the use of a programming language, and there is little agreement in that area, either.

Few computing education programs make a serious attempt to require students to learn a declarative programming paradigm such as functional programming, and even fewer rely on a declarative paradigm at the introductory level. Of the programs that do use a declarative paradigm in CS1 in the United States, Scheme is the most commonly chosen programming language. Only a handful of programs use a strongly typed, higher-order language such as Haskell. The functional paradigm is a more common choice for CS1 in parts of Europe, especially the United Kingdom, but it is not the predominant choice there, either. European programs that introduce the functional paradigm often make use of Haskell-like languages.

This paper argues that the use of Haskell in CS1, rather than a conventional language, can lead to improvements in the education of computing students. While the argument is cast in terms of Haskell, most of the argument applies, with about the same force, to other languages with good support for the declarative paradigm, such as Scheme and ML. In this sense, the title of the paper is misleading. A broader reference to declarative languages, as a whole, as good choices for CS1 would overstate the topic of the paper, but not by much. The paper points out specific advantages of Haskell over alternatives such as Scheme, ML, and Java™.

The argument presented here makes no novel points. But, it puts many known arguments into a unified framework, and that may be of some assistance to educators interested in introducing their students to programming concepts via a language like Haskell.

Further, the paper advocates the use of intrinsic (in Haskell) higher-order operators as the primary means of expressing repetitive computations, avoiding the use of explicit recursion altogether in the programs that novices write. This is an oft-stated goal of functional programmers,

* Address: School of Computer Science, University of Oklahoma, Norman OK 73019, E-mail: page@ou.edu

but one seldom realized in introductory material, judging from the early introduction of recursion in most texts [2] [3] [12]. The text by Bird and Wadler [1] is an exception. Here it is suggested that recursion can be entirely optional in CS1, where the needed computational patterns can be satisfied entirely by a small set of higher order functions. Of course, many instructors regard recursion as an essential element of CS1, but those who do not may devote to other issues the course time that recursion would occupy.

Finally, the paper summarizes experience over the past two years in teaching a CS1 course using Haskell. Coursework has included both individual and team programming projects, made extensive use of a website and email for communicating with students, and, most recently, built on past success by employing especially successful students from previous offerings of the course to staff an email-based help-line.

Convincing ones colleagues to permit Haskell in CS1 is a hard sell. Sharing information on the topic, especially when tied with course experience, may make it easier.

2 Introductory Computing Core

2.1 Current Practices

Most computing curricula have an introductory core containing two or three courses with programming as a central activity. For example, the introductory core of University of Oklahoma's computer science curriculum comprises four courses: *Fundamentals of Computer Programming*, *Programming Structures and Abstractions*, *Discrete Mathematics*, and *Data Structures*. Programming concepts dominate three of these courses, and the other course, *Discrete Mathematics*, discusses some of the underlying mathematical concepts supporting software development. Major goals for students studying the core include learning and practicing several software development concepts: abstraction, program organization, interface specification, documentation, and reaasoning about programs. These are the important issues, and the three core courses with major software development elements focus on them.

After students have acquired a facility for applying these concepts, they can begin to add skills related to detailed control of computing resources. It is a mistake to place beginning students in an environment that forces them to attend to detailed resource management because those matters then require essentially all of their attention, and more important concepts get lost in the muddle.

It is a sad fact that most computer science graduates have had few opportunities to acquire significant experience with the important concepts. One of the most noticeable consequences of this lack of experience is that the most common mode of programming is to create flawed software and then use a symptomatic rather than a systemic approach to repair the flaws. That is, students spend 10% of their time cobbling together codes and 90% of their time fiddling with those codes and observing the effects of their fiddling. They develop the ingrained habits of frantic debugging that permeate the software industry, even though evidence shows these habits to be an order of magnitude less effective than reasoned design and review [6].

2.2 More Effective Practices

The use of programming languages that handle low level details of resource management automatically, such as Haskell and Java, lets novices focus on more important concepts. This helps them look beyond the muddle and acquire effective software development habits from the outset.

They learn to spend 90% of their time thinking (that is, designing their programs and reasoning about the meaning of the code they have written), and they avoid unproductive fiddling. Then, once they have acquired productive habits, they can turn their attention to more primitive programming environments, such as C++, and apply their productive habits in those environments.

It's like learning to drive a car. Students learning on a manual transmission must think of nothing other than clutching and shifting until they have mastered the process of changing gears. Only then can they turn their attention to the more important issues of watching other vehicles, road signs, steering, accelerating, and braking. This is why driving instructors choose cars with automatic transmissions as learning platforms for novices.

For similar reasons, sophisticated programming languages provide the best platforms for introductory courses. With this approach students develop effective programming habits that persist through their later studies, when they may need to use more primitive languages.

3 Reasons to Choose Haskell for CS1

3.1 Programming Languages in Core Courses

The plan advocated here is to use Haskell as the primary notation for introducing the concepts of the first course in the introductory core, and to use Java in the other core courses. For example, at the University of Oklahoma, this would mean using Haskell in *Fundamentals of Computer Programming* and Java in the other two programming courses, *Programming Structures and Abstractions* and *Data Structures*.

There are several characteristics of Haskell that are important in this context. One is that Haskell is an equation-based language without mutable variables. This facilitates reasoning about the meaning of programs. Another is that Haskell manages memory automatically and evaluates only those portions of a program needed to produce the specified results. That is, Haskell programs do not concern themselves with the management of memory or processes, and this leaves the novice programmer free to direct attention to other issues. That Haskell provides a way to control the visibility of entities aids the learning of modular program organization, and static typing helps maintain consistency within the program. (Students often complain about type errors, but usually agree that the alternative — that of running incorrect programs — is worse.) Finally, polymorphism, type variables, and fully composable functions that can process and deliver entities of any type, including arbitrarily complex data structures and function-types, make it possible to treat abstraction much more consistently than in conventional languages.

Java has a similar array of characteristics for memory management, entity visibility, and polymorphism, and it adds mutable variables to the list. Java programs usually describe computations as sequences of states, rather than as equations specifying substitutions that lead to computational progress. Thus, Java programs usually place a greater burden on the programmer concerning resource management issues and reasoning about meaning than Haskell, but draw upon the programmer's attention substantially less in these areas than conventional languages. And, without type variables, Java's support for abstraction may be somewhat less accessible. Haskell's greater sophistication in these areas may have advantages for the introductory course.

The essential feature of this plan is that it follows the strategy of using more sophisticated programming languages early in the educational process to facilitate emphasizing, early on, the most important software issues.

3.2 Why Not Skip Haskell and Use Java Throughout the Core?

Using Haskell in the first course has two advantages over Java: coverage of an alternative paradigm and more effective communication of important concepts. In many computing curricula, the first course in the core is the only required course in the curriculum in which it is practical to introduce an alternative paradigm. Textbooks for CS2 (data structures), compilers, and operating systems based on declarative languages are not available. A curriculum that includes a "tour of programming languages" might devote up to a third of its content to a declarative paradigm, but that is hardly enough to make the point. So, using a language such as Haskell in the first course is the only practical way to ensure that all computing students have significant experience with multiple software paradigms. Emphasis overall remains, as probably it should, on the dominant paradigm of the present period in computing, object oriented programming.

The most important concepts of software development are abstraction, program organization, interfaces between components, documentation, and reasoning about software. These are the concepts the introductory core sequence must communicate. Haskell (and other declarative languages such as Scheme and ML) and Java facilitate focussing on these concepts.

For three of the concepts, abstraction, interfaces, and reasoning, Haskell provides a greater advantage than Java. With regard to discussing program organization and using this concept to facilitate team projects, Haskell has an advantage over Scheme. Instructors who have had students from the Haskell-based version of CS1 in subsequent computing courses at the University of Oklahoma have found that the students' understanding of program organization issues is substantially better than those from conventional courses.

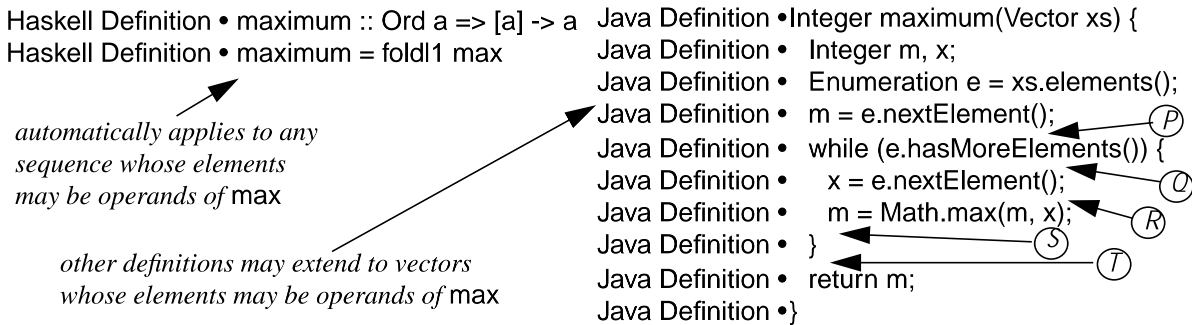
Furthermore, problem solving skills, which most courses in engineering and the mathematical sciences aim to enhance, can be a more significant component of a Haskell-based course than a conventional course. Projects to be completed in Haskell can address more complex problems than projects in conventional languages because Haskell's facilities automate many of the mundane details that conventional languages force the programmer to struggle with. Therefore, more challenging problems fall within the range of programming projects that novice programmers can complete.

With regard to abstraction and interfaces, Haskell prevents software components from interfering with each other. That is, the behavior of one component cannot affect the behavior of another. This isolation of components makes it possible for components expressing abstractions to be based entirely on the interfaces presented by their abstraction parameters. This facilitates the sharing of components between instructor and students and among students working in teams. In both cases, students gain valuable experience with abstraction and interfaces that they can later apply in less constrained environments, such as those of Java or C++.

With regard to reasoning about programs, which is a more effective route to reliable software than testing and debugging [6], Haskell, Scheme, ML, and even Java make the reasoned approach accessible to many students. It is more effective to begin in a declarative language such as Haskell than in Java because in Haskell most reasoning can be based on equations (substituting equivalent expressions for those that appear in the program). Programs amenable to equational reasoning can be written in Java, but most textbooks focus on more conventional methods, where changes in the states of variables render direct substitutions invalid. More delicate logical arguments are required (see Figure 1).

So, in a course based on a declarative language such as Haskell, students can practice reasoning about programs in an environment in which the task is relatively straightforward. This makes

.



The preceding codes define an operation that delivers the largest element from a sequence of comparable objects (integers, for example) in terms of an operation called max that delivers the larger of two comparable objects

Theorem. maximum delivers the largest number in a sequence. That is, the following property holds:

$$\text{maximum}[x_1, x_2, x_3, \ldots, x_n] = \text{maximum}[x_1, x_2, x_3, \ldots, x_n] \text{ `max` } x_i \text{, whenever } 1 \le i \le n$$

proof (Java Definition)

Statement I: m = max(m, y), where y is the current element in the enumeration e or any element preceding it.

Suppose *Statement I* is true at point *Q*. Then at point *R* the current element of e is the next one after the element it was at point *Q*. Because *Statement I* was true at point *Q*, m must be, at point *R*, max(m, y), where y is any element preceding the current element of e. At point *S*, m is the max of the value it had at point *R* and the value of x. Because x is the current element of e at point *S*, *Statement I* is true at point *S* (if it was true at point *Q*).

Any statement that is true at point *S* whenever it was true at point *Q* must also be true at point *T*, provided only that it was true at point *P*. At point *P* the current element of e is its first element because the first application of the method nextElement to e has just occurred. Also at point *P*, m has the value of the current element of e. No elements precede the first element of e. Therefore, *Statement I* is true at point *P*. Consequently, it is also true at point *T*.

Because e has no more elements at point *T* (otherwise, no exit from the while-loop would have occurred) and because m is the value delivered by maximum, *Statement I*, at point *T*, is equivalent to the theorem.

QED *(Java Definition)*

proof (Haskell Definition)

Note: To save space, the symbol ⊕ will stand for the operator ‘max‘.

$$\text{maximum}[x_1, x_2, x_3, \ldots, x_n] \oplus x_i = (\text{foldl1 } (\oplus) [x_1, x_2, x_3, \ldots, x_n]) \oplus x_i \quad \text{— definition of maximum}$$
$$= ((\ldots(\ldots(x_1 \oplus x_2) \oplus \ldots x_i) \oplus \ldots x_{n-1}) \oplus x_n) \oplus x_i \quad \text{— definition of foldl1}$$
$$= ((\ldots(\ldots(x_1 \oplus x_2) \oplus \ldots (x_i \oplus x_i)) \oplus \ldots x_{n-1}) \oplus x_n) \quad \text{— max is commutative and associative}$$
$$= ((\ldots(\ldots(x_1 \oplus x_2) \oplus \ldots x_i) \oplus \ldots x_{n-1}) \oplus x_n) \quad \text{— max is idempotent}$$
$$= (\text{foldl1 } (\oplus) [x_1, x_2, x_3, \ldots, x_n]) \quad \text{— definition of foldl1}$$
$$= \text{maximum}[x_1, x_2, x_3, \ldots, x_n] \quad \text{— definition of maximum}$$

QED *(Haskell Definition)*

Figure 1: reasoning about software — substitution vs. loop invariants

it more likely that they will have a positive experience with reasoning about programs. In subsequent courses, they can extend their practice to environments requiring more subtlety.

An interesting effect of relying on a few specific patterns of repetitive computation rather than ad hoc recursion is that the likelihood of positive early experiences with reasoning about programs is enhanced. Reasoning can use straightforward substitution of new expressions for equivalent ones. In the course experiences on which this paper is based, most students have been able conjure up a proof of the type shown in Figure 1, even under the pressure of a time-constrained examination. After students build confidence from success in this mode of reasoning, more difficult techniques, such as induction, can be introduced.

Finally, Haskell is useful as a specification or design notation, even when it is not being used as the primary programming language in a course. It will, therefore, be useful in other core courses. The primary problem is establishing a schedule of cooperative teaching assignments that will lead to continued use of Haskell as a notational device in project specifications and in explaining concepts. It is important that students see Haskell in subsequent courses. Otherwise, they will resent having been forced to learn something that appears to be of little further use. If students see bits of Haskell in several subsequent courses, they will become accustomed to using Haskell notations in the normal course of software development, regardless of the programming language in which the software is expressed.

3.3 Building Mathematical Maturity and Retaining Students

The early introduction of reasoning about software has additional benefits beyond its influence on programming. For one, it provides a way for computing courses to help build the mathematical maturity of students. Often this important aspect of computing education is largely left to mathematics courses. Juniors exhibit greater mathematical maturity than freshman, partly because they have taken more mathematics, and partly because the students with poor aptitudes for mathematics have failed to make it through the mathematics gauntlet. In any case, computing courses in many curricula play only a small role in developing mathematical maturity.

What is worse, few courses show a connection between software development and the use of mathematics. Students have been left to make this connection on their own. Unfortunately, they do not. Making use of mathematical reasoning in software development requires fundamental insights that only a few students can muster on their own. It also requires considerable practice, which students cannot get if their primary exposure to mathematical reasoning is in courses that have no (or at most a very small) programming component.

Another benefit of building mathematical maturity early in the course of study is that students who do not have the mathematical aptitude necessary to succeed in computing will discover this early in their academic careers. They can then make adjustments in their plans and avoid further delays in completing their degree programs.

This early adjustment will help not only the students who change majors, but also the students who remain in the program, because classes will be smaller. In effect, the department will be able to dedicate more resources to the students most likely to successfully complete their degrees in computing.

Another benefit has to do with attracting good students to computing. Hysterical typing, which accounts for about 90% of the activity of most people who are attempting to write software, does not provide much of an intellectual challenge. Effective programming, on the other hand, is an extremely challenging intellectual pursuit. Dijkstra has called it “one of the most difficult branches of applied mathematics” [4]. Exposing students to this challenge will help attract students who otherwise might pursue disciplines that seem to have more depth.

4 Effects Outside the Computing Curriculum

4.1 Service to Other Programs of the University

Most electrical and computer engineering programs require training in C, and they often learn C in courses from the computing curriculum. When those are service courses for non-computing

majors, no real problem arises. But, if students from other majors take the same introductory computing courses as computing majors, non-computing departments depending on this service are likely to object if C is not one of the languages students learn in the introductory core. This difficult problem must be worked out in each individual situation. Sometimes a separate track is feasible. A few weeks of C near the end of the introductory core is another possibility.

When a minor in computing is offered, the changes have the same effects on students taking a minor as they have on computing majors. So, the analysis of advantages and disadvantages is also the same. Usually, computing minors will have enough background to be able to apply computing to their major area more effectively than students in that area who do not pursue a computing minor. So, the effects on students minoring in computing are more positive than negative.

4.2 Vocational Education

Computing students commonly take internships, other summer jobs, and part time jobs during the school year that require them to know C or C++. It is important for them to have the opportunity to learn those languages early in their programs of study. Part of the mission of most universities is to provide opportunities for this type of vocational training. If resources allow, the department can contribute to this mission by offering one-credit courses in C and/or C++, with some prior training in programming as a prerequisite. At the University of Oklahoma, for example, one way to offer these courses is to make use of the two-week period between semesters. During this period, intensive training courses are offered by some departments. The instructors of the between-semester courses are paid directly by the university, so there is no drain on department's budget. Students can take advantage of such courses to prepare for jobs without having to take an overload during a regular semester.

Many universities have continuing education units that offer courses to the general public (that is, to non-matriculated students as well as regular students). These courses are often self-paced, independent study courses offered for a reasonable fee ($100 to $200), and require a time commitment of 25 to 40 hours for a typical C or C++ training course, assuming some prior exposure to programming. Students could take these courses between the fall and spring semesters to prepare for a second semester part time job, or at the end of spring semester to prepare for a summer job or internship. The required investment is small compared to the benefit. After all, students will be taking this training for the specific purpose of obtaining a paying job.

One may reasonably conclude that the vocational argument against using non-commercial languages in core courses is a red herring. The effects on students are minor, despite the vociferous objections of people opposed to the use of non-commercial languages in computing education. That is not to say the argument is easily overcome. A herring is a potent fish.

5 Experience with Haskell in CS1

5.1 Course Description

Special sections of the CS1 course at the University of Oklahoma have been offered in three semesters, beginning in Fall, 1995. The sections have been small, around twenty students, and the first two of them (Fall, 1995 and Fall, 1996) were offered as part of the university's honors program. Classes have been arranged as two seventy-five minute lectures per week for fifteen weeks: ten weeks using Haskell, then switching to C for the last five weeks to prepare students for their

next software course, which has been using C and C++. (The faculty has recently decided to switch to Java in these courses.) The most recent offering, Spring, 1997, added a weekly fifty-minute discussion session to the class schedule.

The textbook for the Haskell portion of the course [11] emphasizes the concepts of abstraction, program organization, interface specification, and reasoning about programs. Its primary goal is to help people begin to learn to develop software. Programming is the central issue. However, the examples chosen are from important areas of computing, such as representation of information, graphics, sorting/searching, and artificial intelligence.

Students complete seven to nine projects in the Haskell portion of the course, about a third of which are team projects that involve the use of software components developed in previous individual projects. The first project is insubstantial — a logistical exercise to give the students experience in creating files and running Haskell programs. The second project requires the definition of a small function (list-rotation, for example) — different functions for different team members. These components then are assembled into a more complex program in the third project, which is a team effort. The projects continue throughout the course in the pattern or one or two individual projects followed by a team project, and the complexity of the projects increases (display a sequence of strings in a columnar format, prepare a text as a sequence of fully justified lines, normalize and correlate a collection of signals represented as numeric sequences, etc.) so that, by the end of the course, individual projects comprise a hundred or so lines of Haskell.

Each project must conform to an associated style sheet, which expands gradually and stabilizes a few weeks into the course. At that point, project deliverables include a chart showing the modular organization of the program, a literate Haskell script following certain layout guidelines and including type specifications and commentary describing the relationship between the parameters of a function and the result it delivers, a demonstration session with explanations of how input data were chosen and the properties the data were intended to demonstrate, and, for most projects, a proof of a specified property of one of the functions involved in the project. Almost all projects make use of Haskell modules supplied by the instructor.

In this way, the projects direct attention to most of the concepts the course is intended to communicate: program organization, interface specification, documentation, and reasoning about programs. Issues of space/time efficiency receive little attention in the course. Lectures direct attention to abstraction in prepared examples and in reviews of missed opportunities for abstraction in project submissions. Missed opportunities are especially effective because they place the ideas in a context that students have thought intensely about.

This is basically a traditional format for a CS1 course, except for the use of Haskell, although the notion of team projects is another controversial element in some quarters.

5.2 Logistics

Course materials are available via a website [15] for the course, which is accessed an average of eight times per week per student. This includes the syllabus, projects, examinations, frequently asked questions, team rosters, tips on technical issues such as operating system usage, access to an email help line, grades, supplied software, and the textbook. (Hits on the web page containing instructions for downloading the textbook are currently running at about 150 per month, mostly from outside Oklahoma.) Most of these items are also available on paper from other sources, and some are distributed in class. Students have an overwhelmingly positive reaction to the usefulness of the website in their anonymous responses to an evaluation questionnaires that they complete at the end of the course.

One problem in using Haskell for CS1 is a dearth of teaching assistants who know the language. This issue was addressed by teaching the class to honors students for two semesters, thereby building up a cadre of students with the necessary background, then hiring the best of those students to assist with the course. It is unusual to use undergraduate students in this capacity, but the better ones are good at it and are enthusiastic about helping. The arrangement has worked well. The traditional alternative of relying on graduate students would not have worked at all. None of them arrive with the requisite background, and few can afford to invest the effort to get it. So, the success in using talented undergraduates as assistants was fortuitous.

Most of the work of the assistants was in staffing an email-based help line for students in the course. The help line was staffed for a few hours each day, chosen to match well with typical work patterns of students. Requests for help by email arrived at the rate of one-and-a-half to two messages per week per student. About two-thirds of the messages received replies on the same day, and another twenty percent received replies the next day. In most of the remaining cases, replies were received on the third day, but in a few cases it took longer, the longest being six days, which occurred over spring break. Students report that this form of assistance is effective. They sometimes understand their problems better after writing them down, and they can refer back to written replies when problems recur. Student evaluations of help by email is almost as enthusiastic as their appreciation of the website.

The first offering of the course made use of both Hugs [7] [16] and the Glasgow compiler [5]. Subsequent offerings relied on Hugs alone because by that time Hugs included an implementation of modules, a feature needed to support some of the primary themes of the course. The use of a command-response interpreter rather than a compiler makes it possible to focus on software development issues without the distractions of input/output, which is covered in the last few weeks of the course. And, because the compilation stage is abridged, Hugs provides a faster working environment for software development, at least within the range of project complexity occurring in CS1. The automatic module chasing feature of Hugs reduces the number of operating system details that students need to deal with. Finally, it is practical to use Hugs in both Unix® and Windows 95™ environments. Hugs has provided good support for students.

The one negative aspect of Hugs, and this has not been a major problem, has been that students often find it difficult to understand error reports. They learn, eventually, to interpret all error reports as synonyms for the phrase “something is wrong with your program.” This interpretation is less onerous than it seems. In fact it has an unexpected benefit: students are forced to spend more time reviewing the definitions they have made, and the extra time invested in reading and understanding their own programs pays benefits in improvements that otherwise would not have been made.

In addition, baffling error reports have led to some humorous incidents. One student wore to class a tee-shirt with a large semicolon painted on it. When asked the meaning of his garb, he explained that it represented the unexpected semicolon, a reference to the ubiquitous Hugs error message “unexpected semicolon.” (The message is especially confusing for students in this course because they use the offsides layout exclusively — their programs contain no semicolons.) Later, in the C portion of the course, the tee-shirt’s reference became “semicolon expected.” In general, students have been good sports about baffling error messages.

5.3 Team Projects

Students are organized in teams of four to six members. These teams complete a few projects and examinations as a unit. This gives them experience in developing software somewhat more com-

plex than would be reasonable to assign to individuals, and it provides experience in integrating the work of several people into a coherent piece of software. It would be a formidable task to develop team projects in CS1 courses using conventional languages. But, Haskell simplifies the construction of team projects, partly because Haskell provides modules for packaging products of independent efforts, but primarily because individual Haskell components cannot affect the behavior of other components. An intention to include team projects as part of a CS1 offering is a good reason to use a language like Haskell in the course.

Teams also provide a support groups for discussions of course material, and this helps make up for the fact that few students have any experience with Haskell. Students in courses using conventional languages often have many friends who help them get past difficulties, but Haskell is almost universally unknown, at least in the United States, so this kind of help is not available. Not that it would be very good help anyway. Students are probably better off struggling through the material with a few others in the course who are working from the same template. So, team projects have not only engineering-experience benefits, but also learning-assistance benefits.

The experience of this course suggests several important factors in dealing with teams. One is that it is best if they include people of diverse backgrounds and differing skills. Teams must be large enough to make it difficult for a person to get left out (three team members is too few). Teams need something to encourage camaraderie — quizzes taken as a team are one way to develop this. And, teams need to be provided with some sort of structure — specific suggestions for team organization and jobs to do, agenda for meetings, and the like. Most students do not arrive with the necessary interpersonal communication skills and organizational experience needed to work out these details on their own.

Team projects must require discussion, group analysis, and coordination between members. If the project can be done by one person, it will be done by one person. But, projects must small enough to be completed only a few hours of meeting time. It is a mistake to try to see that everyone gets a job to do simply by making the project very large.

Time needs to be provided for teams to meet. Students have busy schedules and find it difficult to get together. (In the most recent offering of the course, the weekly discussion hour, one of three scheduled class periods per week, has often been devoted to team projects.) The instructor needs to observe team meetings frequently and help facilitate solutions when teams run into barriers to progress. Team members staring into space are a common sign of problems. Early mediation keeps such problems within bounds.

All of these ideas for managing team projects proved important in this course. They derive from guidelines formulated by Michaelson from extensive experience in instruction-related teamwork [10].

An evaluation questionnaire completed anonymously by students at the end of the course asked about their degree of agreement with the statement "The team projects improved my ability to do technical work with other engineers and was worth the time I invested in it." Positive and negative responses occurred in about the same number. About half of the positive responses agreed strongly with the statement (as opposed to merely agreeing). No student disagreed strongly with the statement. About ten percent were neutral on the issue. The most common positive comments reported in the evaluation observed that team projects provided an opportunity to work on more complex problems and practice communicating with coworkers on technical issues. The most frequent complaint concerned the difficulty of scheduling team meetings.

5.4 Order of Topics

The order of topics in the course follows that of the textbook [11]. It begins with definitions of simple operations on strings using intrinsic functions such as reverse and equality comparators. List comprehensions are introduced with examples equivalent to mapping and filtering, still operating on strings. It is probably easier to start with numeric examples than with strings, but because computing is mostly about non-numeric computation, it seems truer to the discipline to begin in a domain that has little to do with numbers. Too, numbers in Haskell can lead to some type-errors that confuse novices readily — best to leave numbers until after type specifications have been introduced and students understand more about polymorphism.

Higher order operations provide most of the framework for computational examples in the course. Beginning with composition, other operations are gradually introduced: filtering, folding, mapping, iteration of a function, zipping. The operations give students a way to categorize computations. They ask themselves what is going on at the top level. Is it an assembly line of composed functions? Selecting some of the elements of a sequence? Combining all the elements in a sequence? Applying the same function to all of the elements of a sequence? Or what?

Students seem to be comfortable with using these higher order operations to describe computations. They are not adequate to describe all computations, of course, but they do provide sufficient a range of computing patterns CS1. An examination of sixty exercises in a widely used introductory programming text [8], for example, turned up only five that did not easily fit one of these patterns (plus a few first-order operations on sequences, such as reversing and splitting). And, a pattern of folding and retaining intermediate results handled three of those five. (The remaining two were exercises that involved revising the merge-sort algorithm.) So, it seems that recursion need not play much of a role in a CS1 course using Haskell.

The omission of recursion leaves more time for practice with other programming concepts. Recursion is still needed for input/output in Haskell, but in this context recursion is limited and seems not to require the leap of faith necessary to write recursive functions in a more general context. It is possible to avoid recursion altogether by supplying functions that handle the standard patterns of interactive input/output and file-processing. Students can use these functions as if they were intrinsic operations, needing only an understanding of their effects, not their implementation. But, experience in this course indicates that this measure is not necessary. Recursion for interactive computation presents no more difficulties for novice students than other concepts in the course.

Many educators believe recursion should be a cornerstone in the material of CS1, well worth the investment of course time required for students to come to a solid understanding of it. For those with this outlook, Haskell provides an excellent basis for discussion and illustration, comparable with Scheme, ML, or other languages where the declarative paradigm is commonly used.

Throughout the gradual introduction of higher order operations needed for repetitive computation, the primary concepts of the course, abstraction, interface specification, visibility of entities, program organization, documentation, and reasoning about programs are woven into the examples and repeatedly stressed in lectures, discussion, and projects. The course material skips many aspects of Haskell. Learning Haskell is not a course goal, so the textbook and lectures discuss only a small subset of the language — just enough to illustrate and practice the major concepts.

6 Conclusion

Few computing programs in the United States at the present time offer CS1 using a declarative paradigm as a primary programming tool. Of those that do, most choose Scheme as the programming language. Arguments for a declarative paradigm and more particularly for a strongly-typed, polymorphic, higher order language with automatic memory management, such as Haskell, include the attraction of assigning projects that stress problem solving, focussing on important software issues such as abstraction, program organization, interface specification, and documentation, and encouraging students to reason about their software to avoid falling into the frantic test-and-debug cycle.

Choosing Scheme instead of Haskell would permit about the same level of support for discussing abstraction and reasoning, slightly less support for interface specification, and substantially less support for discussing program organization. The lack of support for packaging entities for reuse has a negative effect on the ease of supplying software for students to use in projects and, especially, on the efficacy of including team projects in the course. ML would provide about the same support as Haskell for all four of the primary concepts. ML does ask more from the novice in terms of understanding flow of control, since ML evaluates subexpressions whether or not their values are accessed at a later point in the computation. It is a convenience not to have to discuss such issues in CS1, but adding them would not be a major distraction for students.

Selling the idea of a declarative paradigm for CS1 requires great patience and long trial periods. Common arguments in opposition include the desire for a balance between vocational training and concept-based education, lack of mathematical maturity in freshman students, and a dearth of qualified student assistants for courses. It helps to be able to point to successful programs because one of the strongest arguments against declarative languages in CS1 is that there is little evidence of success with this approach.

The School of Computer Science at the University of Oklahoma is making slow progress towards the prospect of the general use of a declarative paradigm for CS1. It has been tried in small classes. Plans now include integrating some aspects of Haskell into a large, freshman-level course in discrete mathematics required of all computer science students. If this is successful, the pieces will be in place to make declarative programming a significant part of the introductory core sequence of courses in computing at a public university in the American midwest.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[2] C. Clack, C. Myers, and E. Poon. *Programming with Miranda*™. Prentice Hall, 1995.

[3] A Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[4] E. Dijkstra. *Selected Writings on Computing: a Personal Perspective*, page 129. Springer-Verlag, 1982.

[5] Glasgow Functional Programming Group. The Glasgow Haskell Compiler. www.dcs.gla.ac.uk/fp/software/ghc.

[6] W. Humphrey. *A Discipline for Software Engineering*, page 275. Addison-Wesley, 1995.

[7] M. Jones. Hugs 1.3, The Haskell User's Gofer System: User Manual. Technical Report NOTTCS-TR-96-2, Functional Programming Research Group, Department of Computer Science, University of Nottingham, Nottingham NG7 2RD, England, August 1996.

[8] A. Kelley and I Pohl. *A Book on C*, second edition. Benjamin/Cummings, 1990.

[9] J. Lewis and W. Loftus. *Java Software Solutions: Foundations of Program Design*. Addison-Wesley, 1997.

[10] L. Michaelson. Team learning: a comprehensive approach for harnessing the power of small groups in higher education. In D. Wulff and J. Nyquist, editors, *To Improve the Academy: Resources for Faculty, Instructional and Organizational Development*. New Forums Press, Stillwater, OK, 1992.

[11] R. Page, *Two Dozen Short Lessons in Haskell*. www.cs.ou.edu/cs1323h/textbook/haskell.shtml, 1997.

[12] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley Longman, 1996.

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

[14] A. Tucker, A Bernat, W. Bradley, R. Cupper, and G. Scragg. *Fundamentals of Computing I: Logic, Problem Solving, Programs, and Computers*. McGraw-Hill, 1994.

[15] University of Oklahoma School of Computer Science. *CS1323: Fundamentals of Computer Programming*. www.cs.ou.edu/cs1323h, 1997.

[16] Yale Functional Programming Group.Yale Haskell Project. haskell.systemsz.cs.yale.edu/hugs, 1997.