Functional programming, and where you can put it [1]

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies*; *wadler@research.bell-labs.com*

Rex Page
School of Computer Science, University of Oklahoma, page@ou.edu

While functional programming has played a significant role in a few industrial software projects, it faces barriers to entry that have yet to be overcome. Some barriers, such as performance issues and entrenched resistance to the use of non-standard languages may be less important than it might at first seem, while others, such as a dearth of software developers familiar with the benefits of functional programming, may be crucial factors in many circumstances. One advantage of functional programs is that they are easier to reason about than programs that use mutable variables. This strength makes functional programs an attractive starting point for educators who want to increase the exposure of their students to the practice of applying mathematical logic and reasoning in the software development process. Finding a place in the curriculum for this material is not always easy, but discrete mathematics courses have some advantages that computing educators may want to exploit. The Beseme Project has developed lectures notes, homework projects, examination questions, software tools, and other materials targeting discrete mathematics courses. The materials are freely available and accessible through the internet.

1 Rationale

At a 1993 meeting of IFIP Working Group 2.8 on Functional Programming, I gave a short talk about barriers to the use of functional programming in industry. At the time I was one of about 200 software engineers assigned to a project with the goal of redesigning and implementing most of the software supporting the operations of a large oil company. Plans called for a multi-year effort, deploying components in key areas early and integrating other components as they became available. A few months before my involvement in the project, a team of two developers, with only a few years of experience between them, had chosen the project toolset: a proprietary programming language with built-in support for graphical user interfaces and with a high degree of interoperability with certain SQL engines. To avoid the confusion that might arise from changes in the chosen product since 1993, I will refer to the toolset as Product P. The programming-language component of Product P was a little like C, plus an integrated GUI builder and SQL query generator. Programs were interpreted directly, not compiled to native code.

The choice of Product P surprised me because it ran counter to the usual practices of large corporations, at least as I had observed them up to that time. First, it was a single-source product from a relatively small company only half-a-dozen years old. Second, it was a new development environment, and only one or two of the company's software engineers had previous experience with it.Third, the product was based on an interpreter rather than a compiler, which made software run twenty to thirty times slower than code based on languages used more commonly in the company.

As a foundation for a large investment in software development, accepting the level of risk associated with relying on a small, young company was unprecedented in my experience in large corporations. The training commitment required to go forward with a large project using unfamiliar tools was another strike against Product P that normally would have killed it. The performance issue might or might not, by itself, have prevented the selection of the product, but that was strike three. What kept Product P in the game? More to the point, if Product P could pass muster, why couldn't tools like Haskell or ML?

That was the question I put before WG 2.8. They provided several answers:

- Name recognition — Most people have not heard about Haskell or ML.
- Dearth of programmers — The supply Haskell and ML programmers is small.
- Lack of vendor support — No solid, commercial vendor supports Haskell or ML.

[1]This work is supported by the National Science Foundation (ITR Award 0082849) and the University of Oklahoma.
To appear in *ACM SIGPLAN Notices*, September 2001

- Poor performance — Haskell and ML programs are big and slow.
- Rudimentary tools — Programming environments for Haskell and ML lag.
- Limited resource control — Haskell and ML inhibit direct control of computing resources.
- Restricted side effects — Tweaking system state is not a strong point of Haskell or ML.
- Poor interoperability — Access to databases and GUIs from Haskell or ML is primitive.

Product P scored well on interoperability, good on side effects and resource control, and fair on programming environment. Solving the problem of finding software developers is easier with Product P than with Haskell or ML. Programmers can bridge the gap from well-known languages to Product P in a matter of a few days, since the programming paradigm remains the same. The functional paradigm is substantially different, so getting up to speed on Haskell or ML requires a much greater effort. With respect to performance, name recognition, and vendor support, Product P fares no better than Haskell or ML. If these issues were crucial, Product P could not have been selected.

Prior to this experience, I had assumed that poor performance was the primary liability of functional languages in the marketplace, and the prospect of producing functional language systems that run as fast as C programs is a difficult goal. But, poor performance no longer looms as large, and a solid effort can probably surmount most of the other barriers that functional languages face. Compilers, tools, and interoperability are technical problems whose solutions require a great deal of effort, but they are not intractable. Progress is being made, as can be seen, to cite just a few examples, in the work of Carlsson and Hallgren [5], Rojemo and Runciman [24], Peyton Jones and Hughes [23], Elliott and Hudak [10], Wadler [26], and Leijen and Meijer [19]. Nevertheless it remains true that few software development projects choose functional languages as an implementation vehicle. (Wadler [27] has described some important exceptions, and has also provided some explanations as to why the selection of a functional langauge is the exception and not the rule [28].)

If the problems with compilers, tools, and interoperability were solved, Haskell or ML could stand as good a chance of being selected for future industrial projects as Product P, except for one remaining barrier: the shortage of programmers familiar with functional programming. Removing this barrier calls for sweeping changes in the way software engineers are educated. A few optional courses in the computing curriculum will not make a dent in the problem. If functional programming is to become a tool that most software engineers are equipped to use, it will have to play a significant role in most of the core courses in computing education programs. Without that, students will fail to internalize the usefulness of the functional paradigm, and things will proceed pretty much along the course they have been following for the past fifty years.

2 Starting Points

Sweeping changes in educational processes take a very long time. What might be a reasonable starting point? How about CS1? Why not start students off in a programming language in which they could solve real problems early, instead of having to wade through a mountain of trivialities in the first 80 percent of the course before encountering a project that requires real problem-solving insight? CS1 makes an excellent starting point, and some programs have included functional programming as a primary element in CS1, but the practice is not widespread, especially in the United States.

Courses such as programming languages, software engineering, and discrete mathematics may provide easier opportunities for exposing students to functional programming. When discrete mathematics occurs early in the curriculum, it provides an attractive starting point, especially since it carries little political baggage. Hardly anybody cares about discrete mathematics. Students' eyes glaze over on day one, and they don't wake up until the exam. A few professors get excited about proving the formula for the geometric series or some other result from number theory, but the connection between traditional mathematical examples and software development is subtle, and most students do not see it.

Telling students that they will be better programmers if they know how to use mathematical induction is not convincing, and probably not even true unless they are given a chance to practice using such proof techniques to reason about software artifacts. Figuring out how to apply a theory is at least as big an intellectual hurdle as learning the theory, and most discrete math courses leave it up to students to figure out how to use what they have learned.

The upshot is that the discrete math course, which is part of the curriculum in many computer science programs and is a requirement for accreditation in the United States [7], is a target of opportunity for people who want to make logic and reasoning an integral part of computing education, rather than a peripheral topic studied on its own merits without explicit connections to software development. The discrete mathematics course content is an under utilized resource with regard to such goals, and one that can be co-opted without much gnashing of teeth.

If the goal is to educate students in the use of logic and reasoning on a regular basis in software development projects, functional programs provide a good starting point because their lack of mutable variables simplifies analysis and improves the chances of success. Reasoning about conventional programs, for example using loop invariants, can be introduced at a later stage, when students are more comfortable with mathematical logic and its use in verifying properties of software. (This is not the only approach, of course. Dean and Hinchey [8] edited a collection of papers on the theme of formal methods in the curriculum, and the articles have little to do with functional programming.)

Students following such a regimen will gain some knowledge of functional programming and will be exposed to the idea that logic and reasoning can be an effective way to improve software quality. Of course, one exposure is not enough. Unless the ideas are reinforced at many points in the curriculum, they will not have much impact. But, demonstrated success in this area might facilitate the introduction of reinforcing materials, including both functional programming and logic, in other courses. Such a demonstration is a goal of the Beseme Project, which is described in the following sections. The long-term goal is better software engineering through mathematics education. Increased familiarity with functional programming could be another effect if the Beseme approach succeeds.

3 Strategy

Some of the most highly respected computing scientists, people such as McCarthy [20], Dijkstra [9], Hoare [18], and Backus [1], have demonstrated almost from the beginning, from the early sixties at least and probably before that, the importance of formal, mathematical reasoning in the construction of software. Observation of the use of such methods in practical projects confirms their experience (Selby *et al* [25], Cobb and Mills [6], Gibbs [11]).

Yet, it is a rare software project that makes much use of the methods these prominent thinkers have advocated. Formal, mathematical reasoning is difficult to apply, even for those who know how, and that accounts in part for a lack of common use. However, most students in computing education programs are not required to practice mathematical reasoning in their software projects. Methods absent from education can hardly become standard practice.

Educational materials supporting formal methods in software development (*e.g.*, Gries and Schneider [14], Broda *et al* [4]) have seen some success, but have fallen short of broad acceptance. Despite the endorsement of leading researchers, evidence that integrating mathematical reasoning with programming in computing education leads to improved programming skills is lacking. The Beseme Project seeks such evidence.

For over a decade, the course in discrete mathematics required for the baccalaureate in computer science at the University of Oklahoma has covered Boolean algebra (*e.g.*, and, or, not, DeMorgan's laws), propositional and predicate logic (*e.g.*, implication, negation, contrapositive, modus ponens, quantification, mathematical induction), set theory (*e.g.*, set membership, union, intersection, Cartesian product, relations, functions), and graph theory (*e.g.*, trees, graphs, directed graphs, trees, connectedness, degree), with additional topics depending on the instructor. Examples illustrating these topics are routinely chosen from traditional mathematics. For example, number theory is a favorite hunting ground for examples to illustrate mathematical induction.

However, all of the concepts have many uses in hardware and software development, so examples from these areas could illustrate the ideas at least as well, and the experience of working with these examples might have a positive effect on students' understanding of their primary area of interest, that is computer science or engineering. The Beseme Project has developed a discrete mathematics course with the same, traditional set of core topics, but with revised examples focussing primarily on software development. This course has been offered for the past two semesters (Fall 2000 and Spring 2001) and will continue to be offered in subsequent semesters.

The Beseme approach has not entirely replaced the traditional course, however. There are two sections of the course, and the other section continues in the traditional mode. This produces two populations of students with different experiences in discrete mathematics. In the semester following the discrete math course, most students studying computer science or computer engineering enroll in a data structures course in which most of the activities have a strong programming component. This presents an opportunity to observe differences in the software development skills of the two populations, using performance in the data structures course as an estimator of software development skills. One of the ongoing goals of the Beseme Project is to attempt to detect potential differences of this kind, some of which may be attributable to differing levels of experience in reasoning about software.

4 Products

The Beseme course in discrete mathematics includes a substantial logic emphasis. Most examples relate to reasoning about properties of software. Other topics include trees and graphs (with practical software applications), sets (representation, implementation of basic operations), functions (ordered-pair and formula-based definitions, notions of computability), grammars (phrase generation, language recognition), and so on. Projects and examinations require logical analysis of how programs meet their specifications. Contents of the course, including lecture notes, homework, solutions, reading assignments, examinations, and software tools, are recorded in the project website at http://www.cs.ou.edu/research/beseme.shtml.

Most of the example software is expressed in Haskell. The material must make some notational choices, and Haskell has advantages that make it attractive for the purposes of the project. One, Haskell has similar appearance to algebraic notations commonly used in mathematics, so the overhead of switching between notations is less than it would be with many other choices of language. Two, sequential changes of state in conventional programs create some pitfalls not commonly encountered in traditional mathematical proofs. Haskell programs avoid this source of difficulty by not using mutable variables. These two factors make it easier for students to experience early success in reasoning about software.

Other programming languages could serve equally well, and the materials are adaptable to other environments. For example, the TeachLogic Project based at Rice University (Barland *et al* [2]) is developing similar materials with Scheme as the programming language. The goal of the TeachLogic Project is to teach logic across the curriculum, and discrete math is just one portion of that effort.

Examples verify certain properties of functions defining, for example, reductions on sequences (sum, and, or, length, maximum), sorting, exponentiation, vector addition, sequence reversal, and tree building/searching (AVL trees). Most of the analyzed properties confirm relationships between function arguments and values, but termination and performance properties are also analyzed in some cases. Two lectures focus on reasoning about imperative programs using loop invariants, but the other lectures reason about functions defined in declarative form.

The textbook used in the Beseme section (Hall and O'Donnell [15]) supplies a tool that verifies the correctness of proofs composed in the form of natural deduction. As supplied, the tool requires proofs to be constructed in terms of about a dozen basic inference rules. The Beseme Project enhanced the tool to accept citations of theorems as well as basic inference rules. For the past two semesters, students have used the enhanced version of the tool to confirm the correctness of their natural deduction proofs, and the formality of the proofs in this part of the course seems to increase the confidence of students in carrying out proofs.

A tool for verifying the correctness of equational proofs has been developed and will be used in subsequent offerings of the course. Ongoing work targets the incorporation of predicate calculus and mathematical induction into the proof checking system, and long-term goals aim for a tool that will check proofs of properties of Haskell definitions. The project will not develop tools to automate the creation of any part of a proof, but will concentrate on confirming the correctness of fully detailed proofs composed by students. Students who successfully use tools of this kind may be better able to appreciate the services of more robust theorem-proving systems (*e.g.*, Gordon *et al* [13], Boyer and Moore [3], Hardin *et al* [16], Owre *et al* [21], Gordon and Melham [12], Paulson [22], Heitmeyer *et al* [17]) than students without experience of this kind.

5 Goals

While educational materials and reports on experiences in using them are the primary goals of the Beseme project, its aspirations are broader. The project aspires to influence computing educators in the direction of providing greater emphasis on reasoning about software and, secondarily, to do so in a way that will, over the long term, increase the awareness in the software engineering community of functional programming and the ability of software engineers to use it. Success in this effort might contribute to a small shift in the balance between the test-and-debug cycle on one end of the software development continuum and verification based on mathematical reasoning on the other.

References

[1] John Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of the ACM*, 21(8): 613-641, August 1978

[2] Ian Barland, Matthias Felleisen, Phokion Kolaitis, and Moshe Vardi, TeachLogic Project, http://www.cs.rice.edu/ ian/TeachLogic

[3] R S Boyer and J S Moore, *A Computational Logic Handbook*, Academic Press, 1988

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

[5] Magnus Carlsson and Thomas Hallgren, Fudgets A Graphical User Interface in a Lazy Functional Language, *Proc. Conference on Functional Programming Languages and Computer Architecture*, 321-330, ACM Press, 1993

[6] R H Cobb and H D Mills, Engineering software under statistical control, *IEEE Software*, 7(11): 45-54, November 1990

[7] Computer Science Accreditation Commission, Criteria for Accrediting Programs in Computer Science in the United States, Version 1.0. Computer Science Accreditation Board, January 2000

[8] C Dean and M Hinchey, editors, *Teaching and Learning Formal Methods*, Academic Press, 1996

[9] E W Dijkstra, EWD227 stepwise program construction, 1968. In: *Selected Writings on Computing: a Personal Perspective*, E W Dijkstra, editor, 1-14, Springer-Verlag, 1982

[10] Conan Elliott and Paul Hudak, Functional Reactive Animation, *Proc. 1997 ACM SIGLPLAN International Conference on Functional Programming*, ACM, 1997

[11] W W Gibbs, Software's chronic crisis, *Scientific American*, 86-95, September 1994

[12] M Gordon and T Melham, *Introduction to HOL — A Theorem Proving Environment for Higher Order Logic*, 86-95, Cambridge University Press, 1993

[13] M Gordon, R Milner, R Wadsworth, and P Christopher, Edinburgh LCF: *A Mechanical Logic of Computation, Lecture Notes in Computer Science 78*. Springer-Verlag, 1979

[14] David Gries and Fred Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993

[15] Cordelia Hall and John O'Donnell, *Discrete Mathematics with a Computer*, Springer, 2000

[16] D Hardin, M Wilding, and D Greve, Transforming the theorem prover into a digital design tool: from concept car to off-road vehicle, *Computer-Aided Verification, CAV '98, Lecture Notes in Computer Science 1427*, Alan J Hu and Moshe Y Vardi, editors, 39-44, Springer-Verlag, 1988

[17] C Heitmeyer, R Jeffords, and B Labaw, Automated consistency checking of requirements specifications, *ACM Transactions on Software Engineering and Methodology*, 5(3):231-261, 1996

[18] C A R Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, 12(10):576-583, 1969 (reprinted in Communications of the ACM, 26(1):53-56, 1983)

[19] D Leijen and E Meijer, Domain-Specific Embedded Compilers, *2nd Conference on Domain-Specific Languages*, Usenix, 1999

[20] John McCarthy Towards a mathematical science of computation, *Proc. of IFIP Congress 62*, 21-28, North-Holland, 1963

[21] S Owre, J Rushby, and N Shankar, PVS: A Prototype Verification System, *11th International Conference on Automated Deduction (CADE), Lecture Notes in Artificial Intelligence 607*, 748-752, Deepak Kapur, editor, Springer-Verlag, 1992

[22] L C Paulson, *Isabelle: A Generic Theorem Prover, Lecture Notes on Computer Science 828*, Springer-Verlag, 1994

[23] Simon Peyton Jones and John Hughes, editors, Standard Libraries for Haskell 98, http://haskell.cs.yale.edu/onlinelibrary/, February 1999

[24] Niklas Rojemo and Colin Runciman, Lag, drag, void and use — heap profiling and space-efficient compilation revisited, Proc. ICFP'96, *ACM SIGPLAN Notices*, 31(6):34-41, June 1996

[25] R W Selby, V R Basili, and F T Baker, Cleanroom software development: an empirical evaluation, *IEEE Transactions on Software Engineering*, SE-13(9):1027-1037, September 1987

[26] Philip Wadler, How to declare and imperative. *ACM Computing Surveys*, 29(3):240-263, September 1997

[27] Philip Wadler, An angry half-dozen, *ACM SIGPLAN Notices*, 33(2):25-30, February 1998.

[28] Philip Wadler, Why no one uses functional languages, *ACM SIGPLAN Notices*, 33(8):23-27, August 1998

Rex Page's checkered career has bounced between academic and industrial organizations (including stints in both multinational corporations and Silicon Valley startups) in about equal measure for just over three decades. He is now, as a member of the Computer Science faculty at the University of Oklahoma, working towards the implementation of a baccalaureate in software engineering with a strong emphasis on formal methods and a significant exposure to functional programming.