TeachScheme!—A Checkpoint

Matthias Felleisen
PLT
Northeastern University, Boston, Massachusetts
matthias@ccs.neu.edu

Abstract

In 1995, my team and I decided to create an outreach project that would use our research on functional programming to change the K-12 computer science curriculum. We had two different goals in mind. On the one hand, our curriculum should rely on mathematics to teach programming, and it should exploit programming to teach mathematics. All students—not just those who major in computer science—should benefit. On the other hand, our course should demonstrate that introductory programming can focus on program design, not just a specific syntax. We also wished to create a smooth path from a design-oriented introductory course all the way to courses on large software projects.

My talk presents a checkpoint of our project, starting with our major scientific goal, a comprehensive theory of program design. Our work on this theory progresses through the development of program design courses for all age groups. At this point, we offer curricular materials for middle schools, high schools, three college-level freshman courses, and a junior-level course on constructing large components. We regularly use these materials to train K-12 teachers, after-school volunteers, and college faculty; thus far, we have reached hundreds of instructors, who in turn have dealt with thousands of students in their classrooms.

Categories and Subject Descriptors D.1.0 [*Programming Techniques*]: General; K.3.0 [*Computers and Education*]: General

General Terms Design, Human Factors

Keywords Curriculum Design, Program Design

Designing programs for the benefit of all

The *TeachScheme!* project employs functional programming as a vehicle to deliver two intertwined messages about the introductory programming curriculum. First, if the community wishes to enroll *all* students in a first course on programming and computing, the curriculum must benefit everyone, not just those who continue to program or those who become computer science majors. Second, even the first course on programming should demonstrate that good programming involves a systematic approach, which we call *design*. Simply put, a design-based programming curriculum can benefit everyone.

Our starting point is the insight that programming can easily benefit all students if it aligns itself with K-12 mathematics instruction [2]. Hence functional programming is the most natural fit. Unlike books, functional programming brings mathematics to life for kids—directly and without much ado. In this context, an animation *is* a mathematical function (from time to scenes); an interactive, graphical program *is* a mathematical expression; and a family of web pages *is* the result of some more mathematics. With functional programming, mathematics becomes fun; it is no longer a dry, paper-and-pencil exercise. Best of all, the basic rules of algebraic expression evaluation explain the computational model of functional programming, justifying the idea that it teaches the principles of *computing and programming*. At this point, our curriculum works with algebraic, geometric, trigonometric, and precalculus knowledge; implicitly, it also touches on mathematical integration in several different ways.

Good programming also means planning, organizing, and sticking to a discipline. As such, programming can benefit students by teaching how to solve problems systematically. We realized from the beginning, however, that the connection between conventional programming courses and systematic problem solving was tenuous at best. More commonly, students and teachers would approach programming with the goal of satisfying the parser and getting decent output for a few program runs. Even books on functional programming didn't offer more. If we wanted to use functional programming to teach systematic problem solving, we had to create a curriculum that was explicitly design-oriented. Fifteen years ago, it didn't exist [5].

How to Design Programs (HtDP), our text for high schools and colleges [3], is the principal result of our effort. It uses the ideas of the functional community to teach programming as a systematic activity, i.e., as a design activity. A functional program deals with values; there is no imperative to parse text from some input medium or to write text to some output medium to see how the program works. Values come in a wide variety of flavors: atomic values; compound values; unions; hierarchically nested values; arbitrarily large values; higher-order values; and so on. In sum, functional program design is easy to explain as a two-dimensional grid: one axis describes the process, and the other axis describes the varieties of data. The content of the grid *is* program design, and this grid can be turned into courses for various age groups.

From middle school to college

For the first ten years, we focused on outreach to high schools and on the college-level freshman course [4]. We trained teachers on how to use the first two parts of HtDP; the complete book (and more) was used to teach the freshman course at Rice University. By connecting the two levels explicitly, we hoped that this continuity would guarantee a smooth path into a full-fledged CS curriculum. Over time, however,it became clear that the project needed to expand in two directions: upstream and downstream.

On the upstream side, students encounter computers in middle school (approx. grades 4 through 8), not just high school (ap-

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
ACM 978-1-60558-794-3/10/09.

prox. grades 9 through 12). They listen to their high school friends and siblings when they discuss programming in Python or Java. Conversely, middle school introduces the few algebraic concepts that are needed to understand functional programming. Introducing simple functional programming at this level can help teach essential mathematical ideas, such as *function* and *variable*, while simultaneously preparing the ground for full-fledged programming. It may also preempt students from adopting certain prejudices about programming that we see in so many high school students.

With Emmanuel Schanzer (Harvard U.), we launched the *Bootstrap* project, an after-school program, that is staffed by volunteer teachers. The program currently works with students in some ten underserved neighborhoods across the US. *Bootstrap* provides the strongest evidence yet that teaching functional programming directly affects the mathematics skills and interests of K-12 students.

On the downstream side, students must see how the design principles in HtDP apply to class-based, object-oriented languages such as Java. These languages are what students need for their first co-op or internship. At the same time, these languages do not come with algebraic data types; such forms of data are encoded. These languages do not use pattern matching to evaluate function calls; they rely on method dispatch instead. Last but not least, object-oriented languages support different means for abstracting over repeated patterns; indeed, many abstraction mechanisms are protocols that simulate features built into functional languages.

To establish a bridge between the HtDP course and the mainstream languages that students encounter at work and downstream in conventional curricula, we created a course dubbed *How to Design Classes*. The purpose of the course is to demonstrate how the design principles from the functional world seamlessly apply to the object-oriented world. Next the course moves on to object-oriented means for abstracting code while retaining the design principles for abstraction in functional languages. Finally, the course introduces imperative-style programming with `for` and `while` loops but also explains why doing so violates object-oriented design.

Beyond object-oriented programming, the typical computing curriculum offers two more chances to re-emphasize the messages of design and functional programming: a course on logic and a course on large-scale program development, often called software construction. At Northeastern, we have recently revamped the second-semester logic course [1]. In the past this course employed a rather conventional syllabus, focusing on logic as an exercise in studying formal systems and their meta-theorems. Now the course continues where the HtDP course leaves off—though with ACL2 as the programming language. Students are introduced to ACL2 as an alternative syntax to the teaching languages of the first-semester course. Then they learn to state and prove theorems about their programs. While they start with small functions and theorems, their final project is typically an interactive, graphical game. Students quickly learn that ACL2's theorem prover verifies theorems easily about functions designed according to HtDP and chokes on "spaghetti" functions.[1]

At Northeastern and Northwestern, we have developed a course on *How to Design Systems.* The goal of the course is to remind junior-level students one more time of the design principles of HtDP and to demonstrate how these ideas apply at a large scale. While students choose their favorite programming language to implement a reasonably large system, we demonstrate how the design process of HtDP applies at that scale and to all languages. The key to the course is that the project specification changes on a weekly basis, growing from a short, one-paragraph statement into the description of a distributed system; we also rotate students from one code base to another mid-semester. The students' public design presentations routinely illustrate why a systematic design process is critically important when the project is large and when its specifications continuously change. Students often confirm the importance of this course with notes from their first positions in industry.

[1] Also see Rex Page's *Besseme* project on functional programming in discrete mathematics [8] and on theorem provers in software engineering [9].

Side effects

The "!" in *TeachScheme!* is a pun. One interpretation suggests that the goal of the project is to teach Scheme. It isn't, because the alternative explanation says that "!" is postfix notation for "not." While we never had the intention of teaching plain Scheme, lab observations during our first year drove home the important point that *no off-the-shelf programming language is suitable for novices.* This insight forced us to develop an entire series of teaching programming languages as well as DrScheme, a pedagogical IDE [6], that supports these teaching languages.

The decision to develop our own support software ensured our continued presence in the research community. To support the construction of a series of teaching languages, we developed a programming language for creating full-fledged, ready-to-use programming languages. Over the years, Racket [7], formerly known as PLT Scheme, has served as our platform to explore novel linguistic constructs and to contribute ideas to the functional programming community. In short, *TeachScheme!* created a virtuous cycle—our outreach projects inspire mostly functional research projects, and the results of the research assist our outreach projects.

Acknowledgments Cormac Flanagan asked the right question at the right time; it got us started. Matthew Flatt, Shriram Krishnamurthi, and Bruce Duba immediately agreed to drop everything we were doing and to help launch the *TeachScheme!* project; without them, it would all have been a short daydream. Robby Findler knew what he was getting into when he joined a year later, and he came to build *DrScheme* anyway. Kathi Fisler had the courage to take over my workshops; her contributions have been critical to the survival of the *TeachScheme!* and *Bootstrap* workshops. To all the other members of PLT, thank you very much for your labor of love.

Over the past 15 years, *TeachScheme!* and Bootstrap have received generous support from the Department of Education, the National Science Foundation, Cord, Exxon, Google, Jane Street, and Microsoft.

References

[1] C. Eastlund, D. Vaillancourt, and M. Felleisen. ACL2 for freshmen—first experiences. In *Proc. 7th ACL2 Workshop*, pages 200–211, 2007.

[2] M. Felleisen and S. Krishnamurthi. Why computer science doesn't matter. *Commun. ACM*, 52(7):37–40, 2009.

[3] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004.

[6] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.

[7] M. Flatt and PLT. Reference: Racket. Technical report, PLT Inc., June 2010. `http://racket-lang.org/tr1/`.

[8] R. L. Page. Software is discrete mathematics. In *International Conference on Functional Programming*, pages 79–86, 2003.

[9] R. L. Page, C. Eastlund, and M. Felleisen. Functional programming and theorem proving for undergraduates. In *Functional and Declarative Programming in Education*, pages 21–30, 2008.