

Use Concurrent Programming Models to Motivate Teaching of Programming Languages

Gary T. Leavens
College of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida, 32816
leavens@eecs.ucf.edu

ABSTRACT

Undergraduate computer science students typically have only a limited understanding of their favorite languages and no inkling of other programming paradigms. Yet modern programmers typically work with several languages, and the availability of cheap concurrency is exposing fundamental problems in standard concurrent programming techniques (mutable objects and threads). This situation presents a great opportunity: by exploring nonstandard techniques for gaining intellectual control over concurrent programs, one can motivate and teach important semantic concepts (such as scoping) and important programming concepts (such as functional abstraction). Such a curriculum stimulates student interest in exploring new programming paradigms.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education — curriculum

General Terms

Languages

Keywords

Programming language curriculum, concepts, concurrency, computational models, programming models, paradigms

1. INTRODUCTION

Motivating the study of programming languages is not easy. Most students are not planning to write compilers or design programming languages, and after 2 or 3 years honing their programming skills, they are often unhappy to be forced to study the very tools they have been using all along. Do carpenters study the concepts of hammer design? Wouldn't it be better to get more practice hitting nails?

Many educators also do not believe that undergraduates need to study programming languages extensively. This is evident from the reduced time allocated to programming language concepts in Curriculum 2001 [6]. Thus many schools no longer have a required

course in programming languages and use some of the credit hours formerly given to programming languages to study programming.

This situation is frustrating to those of us who study and teach programming languages. We know that modern languages and paradigms have great power to increase productivity and to encapsulate reusable programming knowledge. How can we motivate the study of programming languages to skeptical students, colleagues, and managers from industry?

My experience is that, while there is no single idea that will magically motivate all students, intellectual control over concurrency¹ is a powerful theme that can motivate and unify a syllabus in programming languages. Thus I advocate using concurrent programming ideas as motivation for teaching programming language paradigms and concepts. In this way a course on programming languages can simultaneously teach programming ideas and programming language ideas, while holding student interest.

In this paper I describe my experience in more detail, describing both my course's design for motivating students and how I strive to meet their heightened expectations. The course, "Programming Languages I" at the University of Central Florida, is a semester long (14 week) course that serves mostly junior and senior undergraduate majors in computer science.

2. MOTIVATING THE STUDY OF PLS

In my experience, many undergraduate students think that a course in programming languages may add particular "useful" languages to their resume. They think the course will teach them practical languages that they may use in programming jobs, such as C and C++, or hone their skills in languages they already know, such as C#, Java, or Python. These are the perceived demands of industry.

Moreover, all the programming techniques my students have seen and all the languages mentioned by their professors use the imperative paradigm. Students also seem to have a sufficient grounding in the basics of imperative languages (assignment, procedures) and object-oriented languages (dynamic dispatch, inheritance, and subtyping), that focusing the course on imperative and object-oriented language mechanisms tends to reinforce their feeling that the main purpose of a programming languages course should be to deepen their understanding of "useful" languages. Thus, when forced to learn a new programming paradigm, they are naturally reluctant, since in the new paradigm they must start over as a beginner.

¹By "concurrency" I mean both shared memory concurrency (task parallelism) as well as data parallelism.

How to motivate such students? How to get them to see the power of other paradigms and the power of programming languages?

I use several motivating arguments, and use concurrency as an overarching theme. Specifically I tell students that:

- Programming languages are changing constantly. Moreover, professional programmers often use several languages in a single application. Both of these factors make it important that students learn how to grasp new languages quickly.
- To grasp new languages quickly, students can identify how the new language relates to standard concepts taught in the course. This will enable them to read a language’s reference manual with greater understanding.
- Programing techniques are closely related to good language design, since both use abstraction to create reusable parts. Indeed many applications have customization features that become more like programming languages over time. Thus the principles of good language design are closely allied to those needed for good program design, and vice versa.
- Cheap concurrent hardware will lead to increasing pressure on programmers to effectively use concurrency. Yet, as many students know from personal experience, writing and debugging concurrent code using threads in an imperative language is fraught with difficulty, due to the presence of race conditions. I explain these problems and promise to show them techniques that avoid these problems.

In my experience, the last point above really sells the course. I have used the first three points in motivating previous programming languages courses, and these were moderately successful. But students still think of the first three points above as perhaps not directly applicable to them. (They have not experienced major language shifts, recall the pain of learning a new language, and have little direct experience with software reuse.)

However, students are very aware of the well-publicized trend in hardware toward multicore computers, and they easily see the problems that race conditions cause for concurrent programs. While they are suspicious of anything deviating from standard techniques that use threads, shared mutable objects, and locking to write concurrent programs, it is not hard to convince them that these standard techniques are far less than ideal. (Indeed many of them have, either in another class or on the job, experienced the problems with such standard techniques for themselves.) Thus, this extra motivation of learning techniques to conquer concurrent programming, when carried through in the course’s design, seems to make a significant difference in student excitement and interest.

3. SYLLABUS

The course is based on an unusual book, which is not designed as a programming languages textbook — *Concepts, Techniques, and Models of Computer Programming* (CTM) by Van Roy and Haridi [7]. CTM uses the Oz programming language, which is a multi-paradigm language with an interesting set of orthogonal features. Although intended as a second course in programming, the book has many features that make it well suited for delivering on the motivating statements described above. In particular it has a large amount of material on concurrency.

The course starts with an overview of different computation models (chapter 1), with a brief overview of each model. In this part of course I emphasize the definition of race conditions and the difficulties it causes for debugging and reasoning about concurrent programs. (There is opportunity here to do more to make these problems clear to students, since Oz has threads and locking, and I plan to do more of this in a future offering of the course.)

The second chapter of CTM introduces the declarative computation model. This model is motivated by promising that its extension, the declarative concurrent model, will avoid race conditions, since it has no mutable storage. The declarative model is defined using a statement-oriented kernel language. It computes over numbers, records (which includes atoms, like Lisp symbols, as a special case), and procedures. Variables in the language have dataflow behavior: once their value is determined (by unification), the value never changes. Attempts to read from an undetermined dataflow variable cause the thread reading the variable to suspend until the value is determined. Statements include unification of variables and expressions, if-then-else, pattern matching **case**, and procedure calls.

While CTM has a formal operational semantics that defines the kernel language, presenting the operational semantics directly is not helpful for my undergraduates (although it seems to work for graduate students). Instead I present the semantics through examples.

The kernel language is very small, but Oz itself has a fair amount of syntactic sugar that makes it easier to use. After explaining the crucial concept of syntactic sugars I spend time teaching the students to desugar from Oz to the kernel language. Many sugars translate expressions and functions into statements and procedures. I also emphasize the concepts of free and bound variable identifier occurrences, and use the desugaring to extend that notion to all of Oz. Since CTM does not treat these concepts at length, I extensively supplement its material, adding an on-line quiz for each concept and more exercises.

The other key concept in the declarative model is referential transparency. I emphasize how important this is for simple (equational) reasoning about declarative programs, and I point out that all programs written in the declarative model (and by extension those that desugar to it) are referentially transparent.

The third chapter covers declarative programming techniques. I emphasize the idea of matching the structure of the data in the structure of the program (“follow the grammar!” [2, 4]). We also discuss how to use examples to derive key steps in a recursive program, and how to abstract away common parts of computations using function arguments and results.

The centerpiece of CTM (its chapter 4) is a discussion of what is called the “declarative concurrent model.” This programming model adds, to the declarative model, threads and lazy execution.² We discuss stream programming and flow control issues, such as buffering. I show how lazy execution allows one to write concurrent demand-driven programs simply. Many of the examples are data parallel programs with a pipelined architecture. The discussion emphasizes that this model is still referentially transparent,

²The authors distinguish lazy execution from lazy evaluation. For them, lazy evaluation uses corouting—when a computation is needed, the thread that needs the value is used to compute the value. In lazy execution, a new thread is used to compute the value.

hence this is concurrent programming without any possibility of race conditions.

Race conditions are necessary, however, to allow client/server programming, where multiple clients can be served without fixing an order in advance. This is the subject of chapter 5 on message passing. The message passing model adds “ports” and a send primitive; all messages sent to a port are merged into a single stream. The main programming technique is to use “port objects,” which encapsulate the port’s stream of messages, thus provide an abstraction similar to an Erlang [1] process. The port object reads from the stream, processing one message at a time. This is similar to but simpler than monitors, as message sending is asynchronous, and so deadlocks cannot directly arise from message sending. Furthermore, declarative concurrent programming techniques can be used inside the port object. In essence, each pair of a message and the object’s current internal state is used to send some messages to other port objects and to generate a new internal state. We also discuss how this model relates to Erlang.

Finally, we discuss the “relational” model briefly. This model is similar to logic programming, but features encapsulated search, which meshes nicely with the declarative models.

I omit extensive coverage of the stateful computation models, including the imperative model and the object-oriented model. While CTM has separate chapters on these models, students already are somewhat familiar with them and I want to focus on stimulating their interest in new paradigms and thus indirectly stimulate their interest in the study of programming languages.

4. RELATED APPROACHES

This section compares the approach discussed above to other approaches I have used in teaching undergraduate programming languages.

I first taught this subject using MacLennan’s book *Principles of Programming Languages* [5]. This book has an interesting set of principles that are used to discuss programming language concepts. However, it does not lend itself to giving students first-hand experience in programming in various paradigms. I was unsatisfied with the shallow level of learning that occurred in such an approach; students seemed to memorize discussion points without really internalizing them. One way to alleviate this problem is to make students write programs in the various languages. However, doing so uses too much time teaching the syntax of each language, dealing with their compilers, etc.

My next experience was with Kamin’s book *Programming Languages: An Interpreter-Based Approach* [3]. This was great for teaching concepts and programming, as students could write programs using various interpreters. All the languages processed by the interpreters were very similar syntactically, which facilitated switching between languages and helped students focus on concepts. However, when students stopped knowing Pascal, the interpreters in the book became less useful.

After that, I used the first and then second edition of *Essentials of Programming Languages* (EOPL) by Friedman, Wand, and Haynes [2]. EOPL is an excellent book that focuses on fundamental semantics of imperative and object-oriented programs. The semantics are explained using interpreters written in Scheme, and in many exercises the students modify these interpreters. Students learn func-

tional programming in Scheme and then use those skills in writing interpreters.³

Most of the functional programming material in EOPL is similar to that in chapter 3 of CTM. The main difference is that Oz has a **case** expression that does pattern matching, which is built into the language and meshes with the record structures used for recursive data (including lists). In my experience, Oz’s **case** statement is a big advantage for teaching recursive programming. It makes the key idea of “follow the grammar!” much more readily apparent in the structure of programs, compared with Scheme.

The other major difference between EOPL and CTM is that my EOPL-based course spends the second half of the semester teaching fundamental semantic concepts from imperative programming languages, while the CTM-based course spends its second half exploring concepts of the declarative concurrent model and the message passing model. Again, this seems to be an advantage for student motivation, as it allows the theme of controlled concurrency to be elaborated in detail. Students also get to see the advantages of concepts like referential transparency in action.

5. DISCUSSION

However, some differences in my CTM-based course may be seen as drawbacks compared with a more traditional syllabus for a programming languages course.

One difference is that there is little place in my syllabus for teaching static type checking. That is, while I explain the distinction between static and dynamic type checking, there is no material on details of how to design or implement a static type system. One difficulty is that Oz is dynamically typed; another is that CTM has little material on this topic. However, it is my experience that students are not interested in the details of static type checking until they have programmed in a dynamically typed language. The experience of programming in a dynamically typed language (like Scheme or Oz) is priceless background for teaching students about static type checking, because it gives them first hand knowledge of the purpose and benefits of static type checking, and the background to understand the tradeoffs involved in a type system’s static approximations. Thus my syllabus may be especially valuable in a curriculum where students are only taught statically typed languages (such as Java, C, C++, and C#).

Another difference compared to traditional programming languages courses is that there is also little place in my syllabus for teaching parameter passing modes. In part this is because it does not make much sense to pass Oz’s dataflow variables by mechanisms such as value-result or result. However, some supplementary material could be added to distinguish call by value, call by name, and lazy evaluation, from the call by reference mechanism used in Oz. In particular a discussion of call by name would be easy to add, since Oz has closures.

Finally, it would be better if more time could be spent having students compare the different approaches to concurrency. That is, I would like students to have more experience directly comparing the standard techniques using shared mutable state and locking with

³EOPL also has material on type checking, continuations, and compilers, but I am not usually able to get to that material in an undergraduate course. In a curriculum where students already know Scheme, it would be easy to cover at least some of the material on type checking.

the declarative concurrent model and the message passing model. There is considerable support in CTM for making such a comparison, but covering 3 more chapters and 200 more pages of the book is beyond what my semester allows. I try instead to emphasize that the techniques of declarative concurrent programming and message passing can be used to supplement such standard techniques. However, this observation reinforces the need for all students to have some first-hand experience with the difficulties of such standard techniques at the beginning of the semester.

In the main, however, it is better to trade some coverage of traditional topics in order to better motivate students and to explore the programming language perspective on concurrency.

6. CONCLUSION

Cheap concurrency is no longer coming to computing, it has arrived. For too long programming language courses have ceded concurrent programming topics to courses on operating systems or systems software. But the programming approaches taught in such courses reflect their low-level focus—they do not (typically) explore alternative paradigms that eliminate or greatly suppress race conditions. We can take back concurrent programming, and by exploring such alternatives in a class on programming languages we can strongly motivate the study of “exotic” paradigms, such as (lazy) functional programming, and message passing. By keeping the students motivated, they are willing to learn the concepts necessary to use these new paradigms, and thus learn about both programming and programming languages.

Acknowledgments

Thanks to Ghaith Haddad who helped developed the initial version of the CTM course at UCF as my teaching assistant, and who gave comments and corrections on an earlier draft. Thanks to the anonymous reviewers from the Programming Languages Curriculum Workshop, whose thoughtful questions stimulated several clarifications and much of the discussion section. Thanks also to the students at UCF whose feedback has helped shape the CTM course. This work was supported in part by a grant from the US National Science Foundation CNS 08-08913.

7. REFERENCES

- [1] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Verdine. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition edition, 1995.
- [2] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, New York, NY, second edition, 2001.
- [3] S. N. Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley Publishing Co., Reading, Mass., 1990.
- [4] G. T. Leavens. Following the grammar. Technical Report CS-TR-07-10b, School of EECS, University of Central Florida, Orlando, FL, 32816-2362, Nov. 2007.
- [5] B. J. MacLennan. *Principles of Programming Languages*. Holt, Rinehart and Winston, New York, NY, second edition, 1987.
- [6] T. J. T. F. on Computing Curricula. Computing curricula 2001. *Journal on Educational Resources in Computing*, pages 1–231, 2001.
- [7] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.