CS3110 Spring 2016 Lecture 1: Introduction, Course Plan, Elements of OCaml: Syntax, Evaluation (1/28)

Robert Constable

Abstract

These preliminary notes contain far more material that was covered in the first lecture. A redacted copy of the notes will be posted next week. We leave these longer notes up now to help students understand better the nature of the course. The actual lecture only covered material up to OCaml syntax.

1 Lecture Outline

2 Course Outline

Goals of the course: functional programming, data structures.

Mechanics:

Schedule
Assignments
Recitations
Exams: Prelim and Final
Academic integrity

Content Summary:

Functions and processes as data
Recursion as control
Data types and type theory
OCaml theory of computing
Asynchronous computation
Analysis of algorithms
    Asymptotic complexity
    Robustness

*Special focus:
    Cyber-physical systems
    Geometric algorithms

3 Course Themes and Mechanics

This course covers elements of *functional programming* in the language OCaml and topics in data structures and algorithms. The OCaml programming language is distinguished by its *rich type system*, and that will also be a focus of the course, connecting to the wider study of type theory in computer science.

Type theory is used in *formal methods* which is an area of computer science concerned with precisely specifying what programs should do and attempting to guarantee that they accomplish the specified tasks. One modern approach to doing this includes using software called *proof assistants* and tools called *model checkers*. I believe that proof assistants will soon be widely used in education and in programming courses like this one. The NSF has just invested another ten million dollars to advance this approach to program correctness. The DoD has invested a great deal more, on the order of one hundred million dollars or more. Microsoft Research is also investing in this area, and Intel has invested for years in using proof assistants to verify elements of its hardware.

Recently DARPA achieved a major success using proof assistants to design an "unhackable drone." You might have seen news coverage of this success. This is an example of work in the area called *Cyber Physical Systems* (CPS). These systems include automobiles and aircraft that use software to help "drive and protect" them. This is currently a very "hot area," and this course will mention one element of this work since some of our researchers have made important advances in this area. The CS Department has identified CPS as a recruiting priority, and last year there were two colloquium lectures in this area, one about drones and one about airplanes.

The large course staff of undergraduates and masters students are highly qualified, e.g. all of the undergraduates on the staff have taken an offering of this course in the past and done very well. The most senior course staff have all done the job before. The current staff will be listed on the web page.

- We require **two recitations per week** for most weeks. We might cancel a recitation now and then in favor of more office hours.
- Only **one prelim** will be given, the date will be listed on the web site.
- There will be **six programming assignments**/problem sets. We will not accept late programming assignments and problem sets. We will deal with medical issues as they arise and may create alternative assignments.
- There will be a final exam as well.

There are previous course notes for about 65-70% of the material in this course, starting with fall 2009 notes. There is an on line resource book, *Introduction to OCaml*, co-authored by a Cornell CS PhD, Jason Hickey, one of my former PhD students. You can freely download it from the course web site. There are other books on OCaml available, some in French.[1]

The OCaml reference manual is also on the web site. It covers the entire language and is a bit dense and terse.

The web page for the course will describe more fully the course mechanics, e.g. programming assignments and problem sets, prelims, quizzes, and a final exam. Please read that material as it appears. It will discuss the role of recitations, consulting, office hours and so forth.

[1]Jason used OCaml to create the*MetaPRL proof assistant* used to produce verified computer programs and to produce formalized mathematics.

The programming assignments are the core of the course. They bring the concepts to life, they teach advanced programming skills, and they allow dedicated students to stand out. We tap the very best students in this course to join the course staff for future offerings. Moreover, the CS faculty recruit student help for their research projects from this course.

Academic Integrity We remind you of the academic integrity policies. If you cheat we will find out, and the consequences will be severe.

4 Course Content

Our course will teach the *OCaml programming language* and how to program in the functional style. You will learn not only this programming language but some new ways to think about the programming process and how to think rigorously about programming languages. These skills will help you understand computer science better, and they might help you get a job in the information technology industry where the ideas we teach are highly valued.

OCaml is a member of the *ML family* of programming languages which includes Standard ML (SML) [12] which we previously used in this course. The family also includes Microsoft's F$^{\#}$, and the original language in this family, Classic ML [4], a very small compact language from 1979 still used in some research projects including mine.

Knowing a language in the ML family is an indication that you were exposed to certain modern computer science ideas that have proven very valuable in writing clean reliable programs and in designing software systems. The ML family is also an excellent basis for presenting the topics in data structures and the analysis of algorithms *because the semantics of the language can be given in a simple mathematically rigorous way as we will illustrate* [6, 9, 5]. This mathematical foundation will allow us to study in some depth the following ideas.

1. Functions as data objects that can be inputs and outputs of other functions, called *higher-order* functions.

2. Recursion as the main control construct and induction as the means of proving properties of programs and data types. Indeed, we will see that *induction and recursion are two sides of the same concept*, an idea connecting proofs and programs in a mathematically strong way [1].

3. We will study *recursive data types* and see that these data types have natural *inductive properties*. We will examine the concept of *co-recursion* and look at co-recursive types such as *streams* and possibly *spreads* as well (trees that can grow indefinitely, also called co-trees).

4. The ML family of functional programming languages is especially appropriate for rigorous thinking about computational mathematics. We will illustrate this by developing some aspects of the real numbers, R in OCaml. These will be *infinite precision computable real numbers*, and they have been used to present in a computational manner most of the calculus you learn in mathematics, science, and engineering [2, 3].

Two or three topics in this course are "cutting edge" in the sense that they are at the frontier of computing theory and type theory. So you will encounter a few ideas that are not typically seen until graduate school in computer science at other universities. These are topics that are especially interesting to the Cornell faculty in programming languages who are known for work in *language based security*.

In particular, we will look briefly at how programs can be *formally specified in logic* and how *proof assistants* can help programmers prove rigorously that programs meet their specifications. We will mention from time to time a particular French proof assistant that is widely used for this purpose, called *Coq*, and its close relative the *Nuprl* proof assistant built and maintained at Cornell. These and other proof assistants (Agda, HOL, Isabelle HOL) have contributed to research in programming languages that are related to OCaml. Coq can generate OCaml code from proofs.

The Coq proof assistant is being used to create a book, *Software Foundations* [14] which formalizes the semantics of programming languages using ideas from the textbook on programming language theory by Pierce [13] and the textbook by Harper [5]. All of the mathematical results in the *Software Foundations* book have been developed with the Coq proof assistant and are correct to the highest standards of mathematics yet achieved because they are mechanically checked by the proof assistant. It is not only that there are no "typos" in the proofs from this book, it is that there are no mistakes in reasoning, and the programs written are completely type correct and also meet their specifications.

OCaml Theory of Computation and Types Every programming language embodies a "mathematical theory of computation". OCaml relies on a *theory of types* to organize its theory of computation. This computation

theory is grounded in sophisticated mathematical concepts originating in *Principia Mathematica* [18] and adapted to programming. For example, you might enjoy reading an extremely influential article by Tony Hoare [7, 11] on data types. After this course you will understand it well.

This version of the course might stress these mathematical ideas a bit more than in the past. We believe that the mathematical ideas underlying OCaml have enduring theoretical and practical value and will become progressively more important in computer science and in computing practice. These ideas are especially important in an age when US cyber infrastructure is increasingly under attack.

This course adds to the functional programming and data structures core other important concepts from computer science theory, namely an understanding of performance, e.g. *asymptotic computational complexity* and an understanding of *program correctness*, how to define it and how to achieve it. We will study methods and tools for organizing large programs and computing systems. We also take up the issue of concurrency and asynchronous distributed computing, a key topic in the study of modern software systems.

Course Topics You can see the sweep of the course and how its main topics are approached from the tentative section headings for our lectures and the accompanying recitations in this offering. They are:

1. Introduction to OCaml functional programming 4 lectures, 4 recitations
2. Data types and structures 6 lectures, 6 recitations
3. Verification and testing 4 lectures, 3 recitations
4. Analysis of algorithms and data structures 4 lectures, 4 recitations
5. Modularity and code libraries 3 lectures, 3 recitations
6. Concurrency and distribution 4 lectures, 4 recitations

These topics account for 25 lectures and there is room for a review lecture and another enrichment lecture. The content is covered in about 25 lectures and 24 recitations.

Special Focus This offering of the course will look at issues in Cyber Physical Systems. One of the new developments arising from the use of proof assistants is that computation is done with *computable real numbers*. These are infinite precision reals which have clean mathematical properties, making it possible to reason precisely about the properties of the code. This is something that is very hard to do when using IEEE floating point numbers which do not have clear mathematical properties.

We will use the computable real numbers to solve problems in computational geometry, e.g. finding the convex hull of a set of points in the plane and finding the area of arbitrary simple polygons. We might also cover the problem of finding all intersecting lines in a region of the plane. These algorithms will be programmed using infinite precision computable real numbers. These are the "real thing." Understanding these numbers will be a significant part of the first half of the course and beyond. We will build up to the reals by using "big nums" for the integers, then defining the rational numbers and finally the real numbers. Learning to compute and reason about these real numbers will be a central focus of several lectures and problems sets.

Lecture Notes and Readings Many of the lecture notes will be from previous offerings of CS3110, however several lectures including this one will be new material and will be posted on the web around the time of the lecture. Some new lecture notes will include the material from previous offerings, perhaps with additions or modifications.

References

[1] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[2] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[3] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

[4] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[5] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, 2013.

[6] Robert Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[7] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[8] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[10] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[11] E. W. Dijkstra O. J. Dahl and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[12] L. C. Paulson. *Standard ML for the Working Programmer*. Cambridge University Press, 1991.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. Electronic, 2011.

[15] Gordon D. Plotkin. LCF considered as a programming language. *Journal of Theoretical Computer Science*, 5:223–255, 1977.

[16] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[17] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[18] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.

[19] G. Winskel. *Formal Semantics of Programming Languages*. MIT Press, Cambridge, 1993.