CS3110s17 Lecture 1:Introduction to Functional Programming with Types

Robert Constable

1 Lecture Outline

1. Course Themes
2. Course Mechanics
3. Syntax of a small OCaml subset
4. Semantics of the small OCaml subset
5. Summary

2 Course Themes

This course is about the *functional programming paradigm and the tasks for which it is appropriate*. To experience the ideas it is critical to know at least one mainstream functional programming language and use it to solve interesting problems. That language should have a rich type system, and we will see why. Basically it is because types provide high value in clearly and precisely specifying programming tasks and in reasoning about program correctness.

On the other hand, if we only provide experience in coding and executing code, students will have an overly narrow understanding of the nature

of computer science and the role of PL (programming languages) in driving the field forward. One of the most remarkable characteristics of computer science is that it is a fast moving field relevant to nearly every other science from mathematics, physics, chemistry, biology, psychology, sociology and so forth. Why is this the case? What major changes will this generation of students witness? Knowing answers to these questions might help you participate in the on-going revolution. We will explore ideas that I find exciting and fundamental. For example, computer science is essential in defining the *ground truth* for all numerical computations. These contributions are the experimental basis of much science and engineering. I don't expect that this will make sense to any of you right now, but at the end of the course, I hope it is a point you will make to others who don't see it.

I mention this example of ground truth because it belongs to the themes that define modern computer science. You need to know the technology, but also know some of the *major research themes*. We will integrate a few of them into this course. I believe this should be part of our upper level CS curriculum.

2.1 The big picture and relevant research themes

For now, Cornell CS has settled on the *OCaml programming language*. The "ml" in the name identifies it as one of the class of languages derived from a programming language designed and built at Edinburgh University [9]. The initials stood for "metalanguage" because it was the language used to define a *Logic of Computable Functions*, LCF for short. Some people now call this small meta language *Classic ML*. We still use ClassicML at Cornell in research. It is a compact simple language. The "o" in OCaml comes from *objects*. This interest reflects the rise of Java and *object oriented languages*. The "C" might appear because some of the influential French computer scientists who designed OCaml as an extension of ML, liked to smoke or liked the name of a certain brand of cigarette. OCaml has served us for several years, so many Cornell students and faculty know this language well. It also has imperative features, and in principle we could use it in that mode, but the point of this course is to stress the functional language features. At some

point the CS Department might adopt another functional language. The programming language of the Coq proof assistant would be an interesting choice, call it CoqPL, and it is being used in some other Ivy League CS departments. We will also refer to it as the "Coq programming language". It is similar to OCaml, and we might use a few examples from that language as well. It will come up later when proof assistants are described.

Another popular language is Haskell, and Scala works well for people who know Java. On the other hand, it is interesting to see a functional language designed to stand on its own because such a language broadens our view of the nature of programming languages in general. We will devote a little time now and then to imagining realistic possibilities for next generation functional programming languages. The Coq PL will be discussed in that context.

Essentially all programming languages (Java, C, C++, etc.) support *functional procedures*. The term "functional language" is typically reserved for programming languages that treat functions as data objects. This means that functions are inputs to other functions, and they are values of other functions. We will have time to illustrate that notion already in this first lecture. Languages like Java, C, C++ and so forth allow "commands" or *imperative features* that change the value of a state. OCaml allows state as well, and it can be treated as an imperative programming language using assignment statements. This version of the course will not stress these imperative features.

The course will cover the functional programming paradigm, algorithm design, and precise problem specification using data structures for which functional languages work well. Typical examples of such data are lists, trees, arbitrarily large natural numbers (Big nums), real numbers, and of course functions. We will also consider data types such as co-lists and co-trees although OCaml does not have these important types. Every programming language must be able to treat numbers well, say integers, $\mathbb{Z}$, and rational numbers, $\mathbb{Q}$. Functional languages are particularly suited to computing with infinite precision real and complex numbers because mathematically such numbers are functions. Functional languages are not typically used to process large matrices.

The OCaml programming language is known for its relatively clean and simple *type system* including *polymorphic types*. Thus types will be a key focus of the course. OCaml types serve as an introduction to the wider role of *type theory* in computer science, not only in programming but in the subject called *formal methods* and in computer security. We will cover a small amount of *software engineering* using the *module* data type and some distributed programming based on OCaml's async package.

Type theory is important in *formal methods* which is an area of computer science concerned with *precisely specifying* a wide range of *computing tasks* and proving that programs exactly satisfy specifications. Types are one way to express what we can say about the structure of inputs to a program and exactly what is required of the outputs as *functions of the inputs*. Logical formulas are also used to make specifications precise. Highly precise *logical specifications* are widely used in mathematics, science, engineering, and commerce. We will see something quite remarkable about OCaml, namely that its type system is rich enough to capture many kinds of logical specifications – without ever mentioning logic explicitly. We will take advantage of this important feature of OCaml types to avoid teaching logic as a separate topic in the course. I find this to be especially significant as well as fun. We might even start on this topic in this very first lecture.

Each scientific and engineering discipline has preferred styles for writing specifications. For example, the language COBOL was designed for expressing "business applications."[1] FORTRAN was designed for numerical mathematics.[2] The richer the type system, the more precisely a programming task can be specified ... up to a point.

The language of mathematics is the richest specification language we know for science, engineering, economics and other disciplines. Its notations and logical specifications are highly precise. For many years school mathematics has been grounded in *set theory*[31].[3] This is a rich yet sparse language that most people interested in programming will have encountered in at least

[1] Can you guess some of the words corresponding to the letters of the acronym?

[2] Can you guess the acronym in this programming language name?

[3] There is a specific set theory, Zermelo-Fraenkel with Choice (ZFC), that is often taken as the foundation for almost all of mathematics[14].

one mathematics course. We will conduct a show of hands to calibrate how common this background is.

SHOW OF HANDS

On the other hand, *there is no mainstream programming language built around the primitives of set theory*, although this has been attempted at NYU. No proof assistant supports the ZFC formalism. I will be asking you why this is true about programming languages and set theory. By the end of the course you will know in detail why. Here is a high level way to think about that question. Set theory was not designed with any computational primitives.[4]

2.2 The concrete example approach

The key construct in a functional programming language is a function, no surprise. In OCaml we write functions in this syntax $fun\ x\ \rightarrow exp(x)$ where $exp(x)$ is an expression that computes to a value when a value is supplied for x. Here is a concrete example $fun\ x\ \rightarrow\ (x + x)$. Functions can be *applied* to values. A clear example is $(fun\ x\ \rightarrow\ (x + x))\ 17$. To compute the value of this application of a function to the number 17, we need to know that $+$ is an operation that reduces $17 + 17$ to the value 34.

Already this simple example raises questions. Is $fun\ y\ \rightarrow (y + y)$ the same function or is it different? Could the variable y in this function already have a value, as in programs with state, e.g. perhaps we assigned the value 17 to y? Does this question make sense?

What about a function defined this way $(fun\ x \rightarrow (fun\ y \rightarrow (x + y)))$? What happens when we apply this to 17? Does OCaml allow us to "assign to the variable y" as in an expression like this? $(fun\ x \rightarrow (fun\ y \rightarrow y := 2; (x + y)))$?

[4]But see SETL [3, 27].

3 Types and type theory

Notice that the sample programs above do not have types on the input variables. We need to figure out what kind of value is supplied by looking at the operations on the value or adding some type information to the variables. You are already familiar with programming languages that provide typing information about the variables, including inputs to functions. It does not work that way in OCaml.

3.1 The story approach to types

Another foundational language for mathematics is type theory, advocated by Sir Bertrand Russell and Albert North Whitehead [29, 26] in a three volume treatise called *Principia Mathematica*.[5] The British computer scientist Sir Tony Hoare showed how to adapt this more expressive language to define the types emerging from the use of programming languages [13]. Well before *Principia*, the Dutch mathematician L.E.J. Brouwer was using an informal computational language to ground mathematics in a rich intuitive notion of computing and a corresponding notion of type that he called a *species*. The type system of OCaml was influenced by these developments in mathematics and logic, and given a *French twist*, perhaps inspired by Poincare's ideas about the foundations of mathematics, related to those of Brouwer [25] whom he influenced.

Functional programs are important because it is possible to reason about them in the way that we reason about functions and types in mathematics. Another way to say this is that functions are a fundamental concept in the mathematics taught in at the university level, starting with calculus and abstract algebra. So in making functions central, we are able to bring to bear on programming a great deal of college level mathematics, including the methods of proving that functions have certain properties. We will rely on college level calculus, algebra, and discrete structures (as in CS2800). **That is why CS2800 is a require prerequisite, not even a co-requisite.**

[5] See the Logicomix [8] story about this book, *An Epic Search for Truth*.

A current popular approach to formal methods uses software systems called *proof assistants* and tools called *model checkers*. The major proof assistants are based on type theory. I believe that proof assistants will soon be widely used in education and in programming courses like this one. The NSF has recently invested ten million dollars to advance this approach to teaching program correctness. One of products of this work is the on-line book *Software Foundations* [22]. The Department of Defence (DoD) is investing a great deal more, on the order of hundreds of millions of dollars to guarantee that critical programs are safe from attack or can respond well to attacks. There are other large investments of which the general public is mostly unaware. Microsoft Research is also investing heavily in this area and is producing very interesting software. One tool is called Danvy that Cornell CS might eventually use for teaching formal methods in Java courses. At the hardware level, Intel and its contractors have invested for many 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 Cornell CS researchers have made significant advances in this area and are keen to help us explain them.

The moderately large CS3110 course staff of highly qualified undergraduates will be helping me teach, manage, and grade for this course. All of the undergraduates on the staff have taken an offering of the course in the past and did very well. The most senior course staff have all done the job before. The current staff will be listed on the web page once it has stabilized.

3.2 The concrete example approach

The key construct in a functional programming language is a function, no surprise. In OCaml we write functions in this syntax $fun\ x\ \rightarrow exp(x)$ where

$exp(x)$ is an expression that computes to a value when a value is supplied for x. Here is a concrete example $fun\ x\ \rightarrow\ (x + x)$. Functions can be *applied* to values. A clear example is $(fun\ x\ \rightarrow\ (x + x))\ 17$. To compute the value of this application of a function to the number 17, we need to know that $+$ is an operation that reduces $17 + 17$ to the value 34.

Already this simple example raises questions. Is $fun\ y\ \rightarrow (y + y)$ the same function or is it different? Could the variable y in this function already have a value, as in programs with state, e.g. perhaps we assigned the value 17 to y? Does this question make sense?

What about a function defined this way $(fun\ x \rightarrow (fun\ y \rightarrow (x + y)))$? What happens when we apply this to 17? Does OCaml allow us to "assign to the variable y" as in an expression like this? $(fun\ x \rightarrow (fun\ y \rightarrow y := 2; (x + y)))$?

4 Course Mechanics

- We offer **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, it will be given **in class** and the date will soon be listed on the web site.
- There will be **six programming assignments**/problem sets. We will not accept late programming assignments and problem sets because of the need to grade them enmasse. We will deal with medical issues as they arise and may create alternative assignments.
- There will be a final exam.
- The final grade will depend about 50% on exams and 50% on the programming assignments.
- Lecture and recitation attendance is not mandatory and will not be monitored, but it is highly advisable to attend and to study the lecture notes. We put a significant amount of effort into these notes.

The statistics over the years support the hypothesis that attendance is valuable to learning and thus to earning a good grade.

There are previous course notes for about 70% of the material in this offering of the 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.[6]

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.

I stress yet again that CS2800 is a **prerequisite** or **co-requisite**. It would be considerably better if students have already taken CS2800 and know that material well since we will draw on it in this version of the course more so than in other offerings. It might be necessary to study in advance some of the CS2800 topics from the textbook for students taking that course concurrently. It is also essential that students know calculus for this version of the course.

Note, this course should not be taken concurrently with CS3410 or CS3420. The work load is too high.

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

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

Academic Integrity We remind you of the Cornell academic integrity policies. If you violate them, it is very likely that we will find out (there are many of us, some extremely vigilant), and the consequences will be severe because we do not tolerate cheating at all.

5 Course Content

This course will teach the *OCaml programming language* and how to program in the functional style. In addition to learning a new programming language, you will learn new ways to understand the programming process better and how to think rigorously about certain features of programming languages. These skills will help you understand computer science better, and they might help you secure 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) [19] which we previously used in this course. The family also includes Microsoft's F$^{\#}$, and the original language in this family, Classic ML [9], a very small compact language from 1979 still used in research projects. We call the original ML [9] *Classic ML*. It is not widely used, but is still alive in some proof assistants.

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* [11, 16, 10]. This mathematical foundation will allow us to study in some depth the following ideas.

1. Functions as *values* (data) that are 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 [2].

3. We will study *recursive data types* and learn 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 [5, 6]. Computable reals provide the **ground truth** for all numerical computation against which IEEE floating point arithmetic can be checked when applications are life critical.

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 from 2004 that is widely used for this purpose, called *Coq* [4], and its older relative the *Nuprl* [1, 7] proof assistant built at Cornell in 1982 and active ever since.[7]

These and other proof assistants (Agda, HOL, Isabelle HOL) have con-

[7]The Coq research team at INRIA won the 2013 ACM system award for Coq, and a key member of the team was Chetan Murthy, a Cornell PhD from the Nuprl group, see Wikipedia, ACM Software System Award, 2013.

tributed 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* [21] which formalizes the semantics of programming languages using ideas from the textbook on programming language theory by Pierce [20] and the textbook by Harper [10]. 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* [29] and adapted to programming. For example, you might enjoy reading an extremely influential article by Tony Hoare [12, 18] 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 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 communication 4 lectures, 4 recitations

These numbers are not exact, but they give an idea of what this course will stress.

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

6 OCaml Syntax

The OCaml alphabet The first step in defining any language precisely, including natural languages, programming languages, and formal logics, is to present its *syntax*. The syntax determines precisely what strings of characters are *programs* and what strings are *data*. The first step is to specify the *alphabet* of symbols used, the "letters of the alphabet of the programming language." Let Σ_{OCaml} be this alphabet. In this section we use Σ for short. We use exactly 94 symbols (tokens, characters) which are the 52 letters of the English alphabet, 26 lower case and 26 upper case, and 32 special symbols from the standard key board, and ten digits, 0 to 9. These are available on standard key boards.

In words the thirty two special symbols are these: exclamation point (!), at-sign (@), pound sign (♯), dollar sign ($), percent sign (asterisk (*), right parenthesis, left parenthesis, underscore, hyphen (-), plus (+), equal (=), right curly bracket, left curly bracket, right brace (]), left brace ([), vertical line (|), colon (:), semicolon (;), quote ("), apostrophe ('), less (¡), greater (¿), comma, period, question mark (?), tilde (∼), backslash, front slash (/), reverse apostrophe (‘).

Latex uses some of these characters to control the type setting, but the names are quite standard. Some have nicknames, such as "squiggle" for tilde.

Hyphen is also a minus sign. The pound sign is sometimes called a "hash", and it is not the sign for the UK currency. Here is a use of square brackets, [...], and here is a use of curly brackets {...}.

The full set of OCaml symbols are from the ISO8859-1 character set with 128 standard characters and 127 others, many are English letters with diacritical marks to spell words in Western European languages, e.g ö, umlaut o. OCaml implementations typically support the standard 94 symbols plus 51 accented letters such as, ö.

Latex shows the need for many more special symbols as does *unicode*. There a many hundreds of special characters that can be printed with Latex and with unicode, and in the future such symbols will be included in the atomic symbols of programming languages. So we might have an alphabet Σ with thousands of letters. Languages like Chinese could show us the limits of comprehension for such rich symbol sets.

OCaml words and expressions Finite strings of the basic symbols we will call expressions or terms. They are an analogue of words in English, even though many are nonsense words, like abkajeky in English. The set of words is denoted Σ^+_{OCaml}, all finite strings of symbols, even nonsense ones such as **1Ab-!. The space (character 0020) is not part of any word in the language nor is a line break or carriage return.

Unlike with natural languages, there is no dictionary of all known OCaml words as there seems to be for (almost) all English or French words. However, there is a dictionary of reserved words such as fun, if, then, else, int, float, char, string, and so forth. There is a largest reserve word (what is it?) but no "largest word" such as "supercalifragilisticexpialidocious" in an OCaml "dictionary", though memory requirements on machines place a practical limit on word length, and in any particular application program there is a list of names of important functions and data types. One can imagine that each project has a dictionary.

OCaml programs and data An OCaml program is simply an OCaml

expression that reduces under the computation rules when applied to a value or given input. *Running a program is evaluating an expression.* A *value* is an OCaml expression that is *irreducible* under the computation rules. We will next look carefully at how to organize the explanation of programs and data. First a word about the scope of this task.

OCaml is a large industrial strength programming language meant to help people do serious work in science, education, business, government and so forth. Like all such languages *it is large, complex, and evolving.* We aim to study a subset that is good for teaching important ideas in computer science. *Thus there are many features of OCaml that we will not cover.* On the other hand, we will present a good framework for learning the entire language as it evolves from year to year as all living languages do.

There is no official OCaml subset for education as has been the case in the past with large commercial programming languages, e.g. at the time when PL1 was a widely used language supported by IBM, there was a Cornell subset called PLC that was widely taught in universities and made Cornell well known in programming languages.[8] The PLC work had an influence on Milners thinking about ML, see the references in *Edinburgh LCF* [9], the first book on ML.

7 OCaml Semantics

A rigorous mathematical method has been developed for precisely defining how programs execute [28, 23, 15, 24]. The concepts are covered in most modern textbooks on programming languages [19, 30, 17, 20, 10]. We will use these ideas to give an account of OCaml semantics. Here is the first key idea of that semantics.

Definition: We divide the OCaml expressions into two classes, the

[8]Many old languages such as COBOL and PL1 are still in use supporting large industrial operations. For mysterious reasons certain languages like C tend to become nearly "immortal". Others like FORTRAN continue to evolve and are immortal in that way. Java, C++, and Lisp might be like that.

canonical expressions and the *non-canonical expressions*. The canonical expressions are the *values* of the language. They are defined as expressions which are *irreducible under the computation rules*. This is a concept that you need to know for exams and discussions. Sometimes we call these values *constants*. This is common terminology for *numerical values* of which OCaml has two types, the *integers* and the *floating point* numbers which are approximations of the infinitary real numbers of mathematics. It is not a word typically used for all of the constants of OCaml, some of which are functions and types.

Exercise: Give five examples of canonical OCaml expressions and five non-canonical ones not mentioned in this lecture.

7.1 Expressions and values

Simple values as constants The integer constants are 0, 1, -1, 2, -2, ... These are constants in decimal notation. They are canonical values because no computation rules reduce them. There is a limit to their size on either 32 bit machines or 64 bit machines. OCaml supports both sizes. Thus these numbers are not like the mathematical integers whose value is unbounded and which thus form an *unbounded type*. OCaml does support an implementation of mathematical integers which in Lisp are called "Bignums."

We may discuss Bignums later, but we will not go into much detail on the limits of OCaml-integers and OCaml-floats. Later in the course we will show how to define *infinite precision* real numbers and thus model the type of mathematical reals R exactly.

The type *bool* is simpler having only two canonical values, the two Booleans, *true* and *false*; simpler still is the *unit type* with one value, ().

Structured values – tuples and records Other canonical forms have structure. For example, (1, 2) is the *ordered pair* of two integers. This is a value, and we call it a constant as well, although unlike the boolean *true*

pairs have structure. OCaml also has n-tuples of values here is a quadruple or four-tuple, (1, 3, 5, 7). OCaml also has values called records which are like tuples, but the components are named as in $\{yr = 2020; mth = 1; day = 20\}$.

Structured values – functions One of the significant distinguishing features of OCaml is that functions are values. They can be supplied as inputs to other functions and produced as output results of computation. Functions have the syntactic form $fun\ x \rightarrow body(x)$, where x is an identifier denoting the input value, and $body(x)$ is an OCaml expression that usually includes x as a subterm, but need not, e.g., $fun\ x \rightarrow 0$ is the constant function with value integer 0. The *identity function* on any data type is $fun\ x \rightarrow x$.

These function expressions are *irreducible*, and thus are canonical expressions. When applied to a value, as in $(fun\ x \rightarrow x)0$ we create a reducible term. In this case it reduces to 0. We see that function values can have considerable internal structure. There is the operator name, fun, an abbreviation of the word function. The identifier x is the *local name* of the input to the function, and $body(x)$ is its "program" or operation on the potential data x.

During computation after an input value v is supplied, this value is substituted for the input variable x resulting in the term $body(v)$. This expression can be canonical or non-canonical. A value is required to initiate the evaluation of a function, but the computation of $body(v)$ might not ever use the value, as in the case of a constant function such as $fun\ x \rightarrow 0$ or $fun\ x \rightarrow (fun\ y \rightarrow y)$.

In the original ML language, now called Classic ML, the function constants have the form $\backslash\, x.body(x)$ which is close to the mathematical notation derived from *Principia Mathematica* and made popular by the American logician Alonzo Church who defined the *lambda calculus* where functions are denoted $\lambda\, x.body(x)$.

There are many notations for functions used in mathematics. In some textbooks we see functions written as in $sine(x)$ or $log(x)$ or even x^2. This notation is ambiguous because we might also use the same expression to

denote "the value of the sine function applied to a variable x."

The programming languages Lisp and Scheme also allow functions as values. Lisp uses the key word *lambda* instead of fun. So $fun\ x \to x + 1$ is written $(lambda(x)(x + 1))$.

As mentioned above one of the other basic syntactic forms of OCaml is the *application* of a function to an argument. This is written as $f\ a$ where f is a function expression and a is another expression. The application operator is implicit in this notation whereas in some programming languages we see application written as $ap(f; a)$ where the operator is explicit.

7.2 Evaluation and reduction rules

The OCaml run time system executes programs that have been compiled into assembly language. This is in a sense the *machine semantics* of OCaml evaluation, but it is too detailed to serve as a mathematical model of computation that we can reason about at a high level. The ML languages have a semantics at a higher level of *reduction rules*. These rules are used in textbooks such as *The Definition of Standard ML* [16].

Evaluation is defined using *reduction rules*. These rules tell us how to take a single step of computation. We use a computation system called *small step* reduction.

Here is an example of a very simple reduction rule. We first note that there are two primitive canonical functions, fst and snd, that operate on ordered pairs, that is on expressions of the form $(e1, e2)$. They are (built-in) primitive operations.

We want a rule format to tell us that $fst(a, b)$ reduces in one step to a and $snd(a, b)$ reduces in one step to b. The rules tell us that we can think of fst as picking out the first element of an ordered pair while snd picks out the second.

Rule-fst $fst(a, b) \downarrow a$
Rule-snd $snd(a, b) \downarrow b$.

Here are rules for the Boolean operators.

Rule Boolean-and $true \;\&\&\; false \downarrow false$
Rule Boolean-or $true \;||\; false \downarrow true$

The general rule for the Boolean operators should take arbitrary expressions, say $exp1$ and $exp2$ and reveal how those values are computed before the principal Boolean operator is computed. To express such rules, we need to state hypotheses about how these expressions are evaluated. Here is the way OCaml performs the reduction.

Rule Boolean-or-1 $exp1 \downarrow true \vdash exp1 \;||\; exp2 \downarrow true$

Rule Boolean-or-2 $exp1 \downarrow false,\; exp2 \downarrow true \vdash exp1 \;||\; exp2 \downarrow true$

Rule Boolean-or-3 $exp1 \downarrow false, exp2 \downarrow false \vdash exp1 \;||\; exp2 \downarrow false$

These Boolean values are used to evaluate conditional expressions.
Rule Conditional-true

$$bexp \downarrow true, exp1 \downarrow v1 \vdash (if\ bexp\ then\ exp1\ else\ exp2) \downarrow v1$$

Exercise: Write the other rule for evaluating the conditional expression.

Here is the rule for evaluating function application.

Function Application

$$exp2 \downarrow v2,\ exp1 \downarrow fun\, x -> body(x), body(v2/x) \downarrow v3 \vdash (exp1\ exp2) \downarrow v3.$$

Notice the *order of evaluation*, we evaluate the argument, $exp2$ first. If that expression has a value, then we evaluate $exp1$ and if that evaluates to a function $fun\, x -> body(x)$, then we substitute the value $v2$ for the variable x in *body* and evaluate that expression. This is called eager evaluation or call by value reduction because we eagerly look for the input to the function, even before we really know that $exp1$ evaluates to a function.

There is another order of evaluation in programming languages where we first evaluate $exp1$ to make sure it is a function, then we substitute $exp2$ in for the variable in the body and only evaluate it if that is required by the body. For example, if the body is just $fun\, x -> x$ then we do not have to evaluate the input first since the body does not "need it yet." This is called *lazy evaluation*. OCaml supports this style of evaluation as well, but we will discuss that later.

These simple rules might seem tedious, but they are the basis for a precise semantics of the language that both people and machines can use to understand programs. By writing down all these rules formally, we create a *shared knowledge base with proof assistants*. It would be very good if OCaml had a complete formal definition of this kind to which we had access. I dont know of one. We could probably crowdsource its creation if we had the ambition and the time.

divergence In all of the evaluation rules for OCaml it is entirely possible that the expression we try to evaluate will diverge, meaning "fail to terminate". That is, the computation runs on forever until memory is exhausted or until you get tired of waiting and stop the evaluation process which is "in a loop." We can write very simple programs that will loop forever without using up memory.

exceptions Expressions might also just "get stuck" as when we try to apply

a number to another number, as in 5 7 or take the first element of a function value, e.g. $fstfun\,x -> (x, x)$. Such attempts to evaluate an expression do not make sense and would get stuck if we tried to evaluate them.

We will see that the type system helps us avoid expressions whose attempted evaluation would get stuck, but we cannot avoid all such situations, and later we will discuss computations that cause exceptions.

Exercise: Write a *diverging computation*, a short non-canonical expression that diverges. This will be discussed in recitation where you will try to find the simplest such expression in OCaml. More subtle question, can there be such an expression that does not consume an unbounded amount of memory?

8 Lecture Summary

Let's go over what we have talked about. This summary illustrates a widely used teaching method. See if you can guess where it is most used. It goes like this: *Tell them what you are going to tell them, tell them, tell them what you told them.*

References

[1] J. L. Bates and Robert L. Constable. Definition of micro-PRL. Technical Report 82–492, Cornell University, Computer Science Department, Ithaca, NY, 1981.

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

[3] Nancy Baxter, Ed Dubinsky, and Gary Levin. *Learning Discrete Mathematics with ISETL.* Springer-Verlag, New York, 1989.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

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

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

[7] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[8] A Doxiadis and C. Papadimitriou. *Logicomix: An Epic Search for Truth*. Ikaros Publications, Greece, 2008.

[9] 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.

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

[11] 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.

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

[13] C. A. R. Hoare. Recursive data structures. *International Comput. Inform. Sci.*, 4(2):105–32, June 1975.

[14] Kenneth Kunen. *Set Theory*. College Publications, 2011.

[15] 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.

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

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

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

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

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

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

[22] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. Electronic, 2013.

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

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

[25] H. Poincare. La logique de l'infini. *Scientia*, 12:1–11, 1912.

[26] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[27] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for the selection of data represntations in SETL programs. *ACM Transactions of Programming Language Systems*, 3(2):126–143, April 1981.

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

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

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

[31] Ernst Zermelo. Untersuchungen über die grundlagen der mengenlehre. *Math Ann*, 65:261–281, 1908.