CS3110 Spring 2016 Lecture 2: Syntax and Semantics (2/2)

Robert Constable

Abstract

1 Lecture Outline

1. Lecture Outline
2. OCaml Syntax
3. Mathematical Semantics for Programming Languages
4. References

2 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 ($\sharp$), dollar sign ($), percent sign (asterisk (*), right parenthesis, left parenthesis, underscore, hyphen (-), plus (+), equal (=), right curly bracket, left curly bracket, right brace (]), left brace ([), vertical line (|), colon (:), semicolon (;), quote ("), apostrophe ('), less (¡), greater (¿), comma, period, question mark (?), tilde ($\sim$), 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 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 $\Sigma^+_{\mathrm{OCaml}}$, 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.[1] The PLC work had an influence on Milners thinking about ML, see the references in *Edinburgh LCF* [1], the first book on ML.

3 Mathematical Semantics for Programming Languages

A rigorous mathematical method has been developed for precisely defining how programs execute [10, 8, 3, 9]. The concepts are covered in most modern textbooks on programming languages [6, 11, 5, 7, 2]. 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 *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

[1] 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.

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.

3.1 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 will use them later in the course we will show how to define *infinite precision* real numbers using big numbers and thus model the type of mathematical reals $\mathbb{R}$ 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 $fun\ x \to body(x)$, 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., $fun\ x \to 0$ is the constant function with value integer 0. The *identity function* on any data type is $fun\ x \to x$.

These function expressions are *irreducible*, and thus are canonical expressions. When applied to a value, as in $(fun\ x \to x)0$ 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 $fun\ x \rightarrow x + 1$ 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.

3.2 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* [4].

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 n step to a and $snd(a, b)$ reduces (in m steps) 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.Note that in OCaml, when we evaluate the pair (a, b) we reduce each of a and b to canonical values before selcting a member of the pair.

Rule-fst $a \downarrow a',\, b \downarrow b' \;\vdash\; fst(a, b) \downarrow a'$
Rule-snd $a \downarrow a',\, b \downarrow b' \;\vdash\; snd(a, b) \downarrow b'$

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.We sometimes use the symbol $\perp$ (called bottom) to denote a *diverging computation*.

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. $fst(fun\, x - > (x, x))$. Such attempts to evaluate an expression do not make sense and would *get stuck* if we tried to evaluate them, and in some cases, "raise an exception" as in $2 * true$ or $1/0$, $2 * [0, 1]$, etc. Some expressions might execute until the runtime system exhausts all memory, as in the recursive procedure **let rec** $loop\, x = loop\, x$.

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?

References

[1] 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.

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

[3] 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.

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

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

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

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

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

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

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

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