CS3110 Spring 2017 Lecture 3: Evaluation Continued and the OCaml Type System

1 Lecture Plan

1. Review and applications of evaluation rules
2. The role of types in understanding evaluation
3. Types: their value and meaning
4. Expressing truths using types
5. OCaml type theory

2 Review of evaluation rules

I know of no complete operational semantics of OCaml. But the account we examined in Lecture 2 and in the other CS3110 resources mentioned are quite informative and manageable. We will use these as a resource for the course, especially the sources accessible from the url's below. People who write compilers for OCaml have this knowledge in some form, and it becomes codified in the OCaml compiler.

`http://www.cs.cornell.edu/courses/cs3110/2015fa/l/13-semantics/core-ocaml.html`

2.1 Applying the operational semantics for an OCaml subset

A new point beyond Lecture 2 that we need to make clear is that expressions in tuples and lists are reduced to their canonical values. So we never have a pair of integers in the form $(2 \times 3, (fun\ x \rightarrow 0)1)$. Instead, the expressions are reduced to canonical form in the pair. The canonical form of the pair is $(6, 0)$. The same principle applies to expressions for values stored in a list, they are reduced to canonical values.

Moreover, in the case of functions as values in a pair or a list, we do not see the canonical form of the functions, e.g. we would not see

$$(fun\ x \to 2, fun\ y \to y \star y)$$

in the list value. Instead we see $[< fun1 >, < fun2 >]$.

```
#  fst ((fun x - > 2*2), 2*2) ;;
>  'a -> int = <fun>
#  snd ((fun x -> 2*2), 2*2) ;;
>  int = 4
```

This unexpected result happens because the functions are *compiled to machine code* for fast execution, and the original expressions are not saved with the data type. In languages with *lazy evaluation*, the values are stored as unevaluated expressions. We will see later in the course how we can mimic lazy evaluation using expressions called *thunks.* These are expressions from the unit type into the value. To recover the value, we need to apply the function to the element of the unit type, $()$.

The needs of the compiler and the runtime system sometimes "out weigh" the mathematical semantics, so we end up seeing some puzzling replies as illustrated in this short session.

```
# (fun x -> x) == (fun y -> y) ;;
- : bool = false
# (fun x -> x) == (fun x -> x) ;;
- : bool = false

# fst ( (fun x -> x), 2*2 );;
- : '_a -> '_a = <fun>

#  fst ((fun x -> 2*2), 2*2) ;;
- : '_a -> int = <fun>
#  snd ((fun x -> 2*2), 2*2) ;;
- : int = 4
```

```
#
================================
# (fun x -> 2*3*x) ;;
- : int -> int = <fun>
# List.hd [(fun x -> 2*3*x); (fun x -> 2*3)] ;;
- : int -> int = <fun>
# [(fun x -> 2*3*x); (fun x -> 2*3)] ;;
- : (int -> int) list = [<fun>; <fun>]
# ((fun x -> 2*3*x), (fun x -> 2*3) ) ;;
- : (int -> int) * ('a -> int) = (<fun>, <fun>)
#
=======================
#  fst ((fun x -> x), 2*2) ;;
- : '_a -> '_a = <fun>
# fst ((fun x -> x/x), 2*2) ;;
- : int -> int = <fun>
# [(fun x -> x/x); (fun y -> 2*3)] ;;
- : (int -> int) list = [<fun>; <fun>]
# List.hd [(fun x -> x/x); (fun y -> 2*3)] ;;
- : int -> int = <fun>
# List.hd [(fun x -> x/x); (fun y -> 2*3)] 4 ;;
- : int = 1
```

Here are some examples that show useful operations on functions as data. It is worth studying these simple examples in detail.

```
fun f -> fun g -> fun x -> f(g(x)) ;;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let h = fun f -> fun g -> fun x -> f(g(x)) ;;
val h : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# h (fun x -> (x+1)) (fun y -> 2*y) 2 ;;
- : int = 5
```

Finally here is an example, like one used in the textbook, that shows how easy it is to write an expression whose evaluation will not terminate, e.g. whose value is Scott's bottom, $\perp$, when we compute this function on any positive integer.

```
# let rec step n = if n=0 then 1 else step (n+1) ;;
```

```
val step : int -> int = <fun>
```

What can we learn from these examples? One striking point is that the top loop evaluator returns type information about the functions. We see this in the line

```
val step : int -> int = <fun>
```

where the expression $int \to int$ gives the type of the function defined in the line above. This type expression means that the object defined in the "let rec" phrase is a function from integers to integers. It is not quite that, it is a function from integers to "partial integers" which includes expressions that diverge when we try to evaluate them.

The type information returned by the OCaml evaluator is extremely useful as we will discover in this lecture. We will see standard types such as integers and booleans and function types like $int \to int$. In addition we will see something quite special to OCaml called *polymorphic types* which are perhaps unique to the ML family of languages. You can play with them by downloading the OCaml evaluator from the web. We will give some very interesting examples in this lecture. They introduce the theme of specifying computing tasks using types, a major theme of this course. The recitations next week will discuss the OCaml evaluator in more detail.

2.2 Expressing facts using the operational semantics

The OCaml language does not provide a vocabulary for stating purely computational equivalence. We might like to write some of the "equalities" we know from the computation rules, for example, how can we say that $(fun\ x \to x + x)7$ reduces to 14 or "equals" 14? In some accounts of programming language semantics, we have a way to say this without carrying out the evaluation, as you can see at the url's included in Lecture 2. Here is another way these equalities are expressed in the literature about programming languages.

$$(fun\ x \to x + x)7 \sim 14.$$

2.3 Currying and Uncurrying

Here is something quite practical about how we can use functions as objects and how they work well with pairs and n-tuples. The textbook covers it as the topic of "currying." An OCaml pair is written (a, b). These are canonical values in that evaluation does not change the outer operator. But note what we saw above in the evaluation of pairs. We select the first and second elements of pairs using the operators fst and snd. We can define a function called *curry* that converts a function whose inputs are pairs or tuples and produces another function whose inputs are individual values. Here is its definition

$$fun\ p \to (fun\ x \to (fun\ y \to\ p(x, y))).$$

```
# let curry = fun p -> (fun x -> (fun y -> p(x,y))) ;;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# curry (fun z -> fst(z)+snd(z)) 5 7;;
- : int = 12
```

We can also define a function to "do the opposite" called *uncurry.* We will discuss this in the future.

3 OCaml Types and Coq Types

The on-line textbook for the course, *Real World OCaml* provides a very readable account of the basic OCaml types in Chapter 1, A Guided Tour. It is recommended reading for this lecture. *Chapter 3 on Lists and Patterns* is also recommended and need for the first problem set. Also *Chapter 6 on Variants* covers one of the novel types of OCaml and a signature feature of OCaml, the *matching* operator on variants. The same matching style is used for lists and can be applied to other constructors as well such as pairs.

OCaml and the whole ML family of languages, Classic ML [3], Standard ML of New Jersey (SML) [4, 9, 10, 5], and OCaml [11, 6, 7, 8] provide rich *type systems.* This distinguishes them from the Lisp and Scheme families of programming languages which do not support a rich type system. The OCaml type system and the others used in standard programming languages are not yet sufficiently expressive to support the full range of mathematics as is the type system of the *Coq proof assistant* [1, 2] whose programming language is very similar to OCaml. So we can view the OCaml type system as an introduction to a type system that is adequate to describe most computational ideas you will see in any college mathematics course. The main extension provided by Coq is the use of *dependent types* and the encoding of logical operators, including quantifiers, into these types. We will explore these types later in the course and use them in precisely specifying programming tasks.

In addition, Coq requires that *functions are total*, i.e. they halt on all inputs. OCaml functions are *partial* computable functions, meaning that on some inputs the computation might fail to terminate. We see more examples of this in Problem Set 1. Types are called *partial* when they permit elements whose evaluation might fail to terminate. As we mentioned before, we sometimes write Scott's bottom, $\perp$ to represent the result of a nonterminating computation. That is, for partial types as in OCaml, some expressions that the type checker allows will *diverge* (fail to terminate) when we attempt to evaluate them. The OCaml type system is based on these *partial types.*

Atomic types The OCaml *atomic types* include: bool, int, float, char, string. We look first at the **integers** int, $\mathbb{Z}$, which is one of the most basic types for mathematics. You might have heard of Leopold Kronecker in this regard. He was a distinguished German mathematician in the 1800s when mathematics was going through a crisis that required being more rigorous to avoid contradictions arising from dealing with "the infinite," as in summing an infinite series and using an infinite series to define functions.

He investigated number theory, algebra and logic. He also criticized Cantor's work on set theory, and was quoted as having said at an after dinner speech: "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk" ("God made the integers, all else is people's work."). He was protesting that mathematics should not be grounded in set theory but rather in an understanding of how to build up mathematical objects using the integers. We can clearly code the booleans as integers, say 0 and 1. We will see how to define real numbers later in the course, a vast enrichment of float. We can encode characters and strings as integers if we want to reduce everything to numbers – that's what happens inside the implementation of this non-numerical data. There were other distinguished mathematicians who were sympathetic to Kronecker's views, such as Poincare, Borel, Weil, and Brouwer, and we will see that they have contributed in fundamental ways to computational mathematics and to computer science before it was a separate discipline from mathematics.

In mathematics we are accustomed to thinking about sets of numbers, such as the set of *natural numbers*, $\mathbb{N}$, say $0, 1, 2, ..$ and the set of integers $0, 1, -1, 2, -2, 3, -3, ...$ What is the difference between the *set of integers* and the *type of integers*? The key difference is that the type provides data and operations for computing with that data based on *computation rules*, and sets do not. Indeed, in a set theory course or even a calculus course, the natural numbers are defined in terms of sets and are simply special kinds of sets, namely 0 is the empty set ϕ and 1 is the set whose only element is the empty set, $\{\phi\}$, and two is the set $\{\phi, \{\phi\}\}$, and so forth. So the number two is actually the set $\{0, 1\}$ and three is the set $\{0, 1, 2\}$. This is all very elegant, and the operations of addition, subtraction, multiplication and so forth are defined as *relations*, that is, sets of ordered pairs. *There is no computation in sight in set theory.* But there are logical rules that allow us to rigorously prove properties of the arithmetic operations, e.g. that $n + m \;=\; m + n$, and other algebraic laws. For example, we can check that $5 + 2 \;==\; 7$.

In OCaml, we know how to compute with the numbers, but the computations are not typically related to the reasons that we know certain mathematical facts. *We will see in this course that there is a very tight connection between computation and mathematical truth.* The way Brouwer explained this is that *computations are a way to experience mathematical truths.* He is famous for saying that *there are no non-experienced truths in mathematics.* That belief connects a certain style of mathematics directly to computer science and to programming, especially to functional programming. That style of mathematics is called *constructive mathematics* or computational mathematics. Why is that? It is because so much of mathematics deals with functions. This is a theme we will stress in the course to enrich our understanding of functional programming. *It will also help us understand how to know that our programs are correct.* This is also the kind of truth we "can experience" once we understand types and see how they relate to proving mathematical properties of programs and data types.

The OCaml type system provides a way to reason precisely about computation. For example, we can say that for any expressions n, m for which we know n is of type *int* and m is of type *int*, then

$n + m, n \star m, n - m$ are all of type *int*. This is a general fact about computation that can be expressed cleanly using types. Keeping track of types helps us avoid attempting to evaluate expressions that do not make sense, such as $2 :: 3 \; = \; 5$. It helps us create expressions that won't *get stuck* when we try to evaluate them. On the other hand, types won't do everything we want. Notice that it will type certain non-terminating expressions as integers. This means that the OCaml type of integers is not exactly like the corresponding mathematical type. We call the type *int* a *partial type* because expressions whose computations do not terminate at all are classified as mapping *int* to *int*.

Can we add to OCaml a static check that expressions will converge to numbers if they have type *int*? What can we say about values of loop, say, *loop* 5?

```
# let rec loop n = if n = 0 then 1 else loop n+1 ;;
val loop : int -> int = <fun>
```

The type of **bool** for Booleans is one of the simplest types, having only *true* and *false* as members. Is this type also partial? That is, can there be a nonterminating expression whose type is a Boolean if it terminates? There are also primitive Boolean operations which we have discussed in Lecture 2 on the computation system. What about the type of **Characters**, is it partial as well? The other key atomic type are the **string**. The type **unit** has only one canonical element. Is it also partial?

Compound Types We have already looked at the type of functions $int \rightarrow int$, the type of OCaml (partial) computable functions from integers to integers. We know that the canonical forms $fun\ x \;\rightarrow\; b(x)$ have this type if $b(x)$ is of type *int* when the input x is of type *int*. We can also write the functions with the "let construct" as follows

```
# let f x = (x * x) ;;
```

In this form OCaml can infer the type of x to be *int* because it knows the type of integer multiplication. In general if $b(x)$ has operators that require a specific type, such as *int*, then the type can be *inferred*. We can also force this typing by explicitly typing the input and output. Here is the example used in the textbook on page 7.

```
# let sum_if_true (test: int -> bool) (x:int) (y:int): int =
(if test x then x else 0) + (if test y then y else 0) ;;
val sum_if_true: (int -> bool) -> int -> int -> int = <fun>
#
```

This situation is general for inferring other such typings as well.

Polymorphic functions Some functions can have many types. For example, $fun\ x\ \rightarrow\ x$ is the identity function on *any type*. OCaml can say this using *type variables* of the form $'a$. There are many examples of such polymorphism. Here is another one: $fun\ x\ \rightarrow\ (x, x)$. What is its type?

In printed documents, the polymorphic types are sometimes written with Greek letters. So the identity functions can have the type $\alpha \rightarrow \alpha$ and polymorphic ordered pairs have type $\alpha \times \beta$. So we know that $funp \rightarrow fstp$ has type $(\alpha \times \beta) \rightarrow \alpha$. How can we see this as an experienced truth?

```
# fst (1,2) ;;
- : int = 1
# fun x -> fst x ;;
- : 'a * 'b -> 'a = <fun>
```

References

[1] 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.

[2] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning (JFR)*, 3(2):1–93, 2010.

[3] 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.

[4] 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.

[5] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions of Programming Language Systems*, 15(2):211–252, April 1993.

[6] Xavier Leroy. *The Caml Light System, release 0.6: Documentation and User's Manual*, September 1993.

[7] Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.

[8] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2002. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

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

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

[11] Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, Paris, 2nd edition, 1999. In French.