CS3110 Spring 2016 Lecture 3: The OCaml Type System

1 Lecture Plan

1. Review of evaluation rules
2. Comments on lazy versus eager evaluation
3. Expressing truths in the evaluation system
4. Types: their value and meaning
5. OCaml type theory

2 Review of evaluation rules

There is no complete operational semantics of OCaml that I know about, either big step or small step. But the account we studied in Lecture 2 is quite informative and manageable. We will take this to be a resource for the course:

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

2.1 An operational small step semantics for an OCaml subset

One point 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 $(2 \times 3, f(5))$ for some defined function f. Instead, the expressions are reduced to canonical form in the pair. If $f(5) = 17$, then the canonical form of the pair is $(6, 17)$. The same principle applies to expressions for values stored in a list, they are reduced to canonical values.

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

$$[fun\ x \rightarrow 2, fun\ y \rightarrow y \times y]$$

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 is 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 "trump" 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) ;;
Characters 4-21:
  fst (fun x -> x, 2*2) ;;
      ^^^^^^^^^^^^^^^^^
Error: This expression should not be a function, the expected type is
'a * 'b
```

```
#  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
```

2.2 Expressing truths with the operational semantics

3 OCaml Types

The on-line textbook for the course, *Real World OCaml* provides a very readable account of the basic 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 well one of the novel types of OCaml and the signature feature of matching on variants. The same matching style is used for lists.

`https://realworldocaml.org/v1/en/html/a-guided-tour.html`

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. These type systems are not rich enough to support the full range of mathematics as is the type system of the Coq proof assistant [1, 2] whose programming language is basically OCaml. A 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. 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. Types are called *partial* when they permit elements whose evaluation might fail to terminate. That is, for partial types, some expressions that the type checker allows in the type might diverge (fail to terminate) when we attempt to evaluate them.

Atomic types The OCaml *atomic types* include: bool, int, float, char, string. We look first at the **integers**, int, which is one of the most basic types for mathematics.

In mathematics we are accustomed to thinking about sets of numbers, such as the set of *natural numbers*, 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 comes with *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 we don't know how to prove much about their mathematical properties. The type system provides a way to say more about computation. 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 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 nothing like the 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*?

```
# 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 $(\alpha \times \beta) \rightarrow \alpha$.

```
# 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.