Rules of Evaluation of OCaml Expressions

P. Madhusudan

September 2018

1 A fragment of Ocaml

Expressions: Let us define a fragment of Ocaml for which we can write formal rules of evaluation.

$$e ::= c \mid x \mid (e_1, ..., e_n) \mid op\ e \mid e_1\ e_2 \mid fun\ x \rightarrow e \mid let\ (x_1, \ldots x_n) = (e_1, \ldots, e_n)\ in\ e$$

$$\mid if\ e\ then\ e_1\ else\ e_2$$

$$\mid \quad match\ e_0\ with\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$$

In the above, e, e_i denote an expressions. c denotes constants, i.e., concrete values in various datatypes (like 23, 3.1415, "bilbo", etc.). *op* is a built-in primitive operator (like + or +. or $\hat{}$), interpreted as being curried and hence taking only one parameter at a time. x and f are identifiers, and p_i are patterns.

Note that 1-tuples (e_1) are synonymous with e_1. Hence "$let\ x_1 = e_1\ in\ e$" is written really as "$let\ (x_1) = (e_1)\ in\ e$", and we avoid a special syntax for it.

Note that the above fragment disallows recursion ("let rec "), algebraic datatypes, etc. It also does not allow pattern matching in "let" statements (this can be accommodated easily, though).

2 Evaluation

Values Let us define the set of *values* (*Values*) as a subset of expressions (as defined above) and *closures*:

$$v, v_i ::= c \mid (v_1, \ldots, v_n) \mid \langle x \rightarrow e, \rho \rangle$$

Environment Let us define environments as partial functions $\rho : Ids \rightarrow Values$. Environments have finite domains.

Let us also define *updates* on environments. Given two environments ρ_1 and ρ_2, with domains D_1 and D_2 respectively, the domain of $\rho_1 + \rho_2$ is $D_1 \cup D_2$, and for each literal $x \in D_1 \cup D_2$,

$$\begin{aligned}(\rho_1 + \rho_2)(x) &= \rho_1(x)\ \ if\ x \in D_1 \\ &= \rho_2(x)\ \ otherwise\ \ \ i.e.,\ if\ x \in D_2 \setminus D_1\end{aligned}$$

The above says that the environment ρ_2 is updated by ρ_1's definitions, and the definitions given in ρ_1 supersede those in ρ_2.

Evaluation We can now define the evaluation function *Eval* that maps pairs, each pair consisting of an expression and an environment, to a value. The notion of an evaluation, strictly speaking, will be a partial map in general (evaluation will not be defined if for example an expression uses a variable that is not defined in the environment, or if when we have recursive functions, the evaluation of the functions do not terminate).

We will also define a function *App* that evaluates a closure on an expression to a value. These two functions, *Eval* and *App* are mutually recursively defined.

These recursive definitions will be defined using *equations*. Unlike many recursive definitions you typically write, there is no *measure* that decreases (i.e., when defining the evaluation of an expression, the definition may involve a recursive definition that could be on *larger* expressions). This does not happen in the definitions below, as we do not handle evaluation of recursive functions, but they will happen in general. The semantics of these equations is hence defined using what is called *least-fixed point semantics*— we mean that the *least* partial map that satisfies the equations is the meaning of *Eval* and *App*. You may wonder why such a least map must exists— it can be shown to be exist using Tarski-Knaster theorem, which says the least fixpont exists because the right-hand-side of these definitions are *monotonic*. Let's not worry about these for now.

$$
\begin{aligned}
\mathit{Eval}(c,\rho) &= c \\
\mathit{Eval}(x,\rho) &= \rho(x), \quad \mathit{if}\ \rho(x)\ \mathit{is\ defined} \\
&= \mathit{undefined}, \mathit{otherwise} \\
\mathit{Eval}((e_1,\ldots,e_n),\rho) &= (v_1,\ldots,v_n),\ \mathit{where}\ \mathit{Eval}(e_n,\rho)=v_n,\ldots,\mathit{Eval}(e_1,\rho)=v_1 \\
\mathit{Eval}(\mathit{op}\ e,\rho) &= [|\mathit{op}|](v),\ \mathit{where}\ \mathit{Eval}(e)=v \\
\mathit{Eval}(e_1\ e_2,\rho) &= \mathit{App}(\mathit{Eval}(e_1,\rho),\mathit{Eval}(e_2,\rho)) \\
\mathit{Eval}(fun\ x \to e,\rho) &= \langle x \to e,\rho\rangle \\
\mathit{Eval}(\mathit{let}\ (x_1,\ldots,x_n)=(e_1,\ldots,e_n)\ \mathit{in}\ e,\rho) &= \mathit{Eval}(e,\{x_1 \to v_1,\ldots x_n \to v_n\}+\rho) \\
&\quad \mathit{where\ for\ each}\ i, \mathit{Eval}(e_i,\rho)=v_i \\
\mathit{Eval}(\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2,\rho) &= \mathit{Eval}(e_1,\rho) \quad \mathit{if}\ \mathit{Eval}(e,\rho)=\mathit{true} \\
&= \mathit{Eval}(e_2,\rho) \quad \mathit{if}\ \mathit{Eval}(e,\rho)=\mathit{false} \\
\mathit{Eval}(\mathit{match}\ e\ \mathit{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n,\rho) &= \mathit{Eval}(e_1,b_1+\rho) \quad \mathit{if}\ \mathit{Eval}(e,\rho)\ \mathit{matches}\ p_1\ \mathit{producing\ binding}\ b_1 \\
&= \ldots \\
&= \mathit{Eval}(e_n,b_n+\rho) \quad \mathit{if}\ \mathit{Eval}(e,\rho)\ \mathit{matches}\ p_n\ \mathit{producing\ binding}\ b_n \\
&= \mathit{undefined} \qquad \mathit{otherwise}
\end{aligned}
$$

$$
\mathit{App}(\langle (x_1,\ldots x_m) \to e,\rho\rangle,(v_1,\ldots,v_m)) = \mathit{Eval}(e,\{x_1 \to v_1,\ldots x_m \to v_m\}+\rho)
$$

Some remarks on the above definitions, line by line:

- Constants evaluate to themselves.
- A variable evaluates to the value given to it by the environment.
- A tuple of expressions evaluates to a tuple of values obtained by evaluating the individual expressions. Note that we don't specify above the order in which these expressions are evaluated– they are typically evaluated right-to-left, or the order may be unspecified, in which case we assume compilers can evaluate them in any order and the programmer cannot assume a particular order.
- A primitive operation *op* is evaluated by calling a pre-existing implementation of the function associated with *op*, which we call $[|op|]$.

- When evaluating a function application (e_1 e_2), we first evaluate e_1 (to get a closure) and evaluate e_2, using the current environment, and then call *App* on the resulting values. Note that *App* itself does not get the environment ρ (it will have access to the environment in the closure for evaluating the body of the function, but does not need the current environment ρ).
- Evaluation of an anonymous function definition evaluates to its closure, where the closure imbibes the current environment.
- In evaluating a $let\ (x_1, \ldots, x_n) = (e_1, \ldots e_n)\ in\ e$ expression, we first evaluate the expressions $e_1, \ldots, e_n$ (in some order, but Ocaml usually does this in reverse order) and then evaluate the expression e in the environment that has been updated with the bindings of each x_i to the corresponding value.
- In evaluating a conditional, the condition e is first evaluated. Assuming this evaluation is well defined (terminates), we evaluate e_1 if the condition evaluates to true and evaluate e_2 if the condition evaluates to false. Note that if the condition evaluates to true, e_2 in *not* evaluated and, similarly, e_1 is not evaluated if e evaluates to false.
- In evaluating a match statement, we evaluate e_0 first, and then find the first pattern p_i that matches and find the binding that it results in. We then evaluate (only) the expression e_i corresponding to the pattern p_i, with the environment updated with the binding b_i, We don't describe here formally how matching is done.
- The *App* function takes a closure and a tuple of values and applies the function body in the closure by updating the environment in the closure (which is the environment at the time of creation of the function) with the given values for the parameter to the function.