COSE212: Programming Languages

Lecture 5 — Expressions (1)

Hakjoo Oh

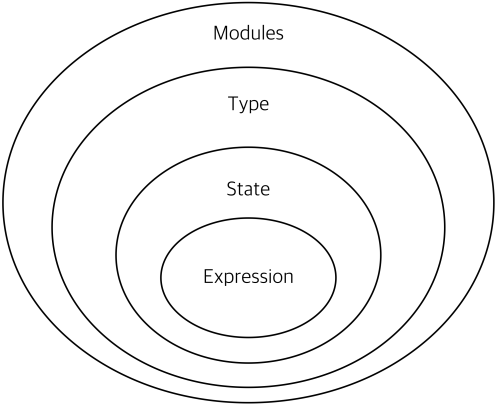
2015 Fall

Plan

- **Part 1 (Preliminaries):** inductive definition, basics of OCaml programming, recursive and higher-order programming
- **Part 2 (Basic concepts):** syntax, semantics, naming, binding, scoping, environment, interpreters, states, side-effects, store, reference, mutable variables, parameter passing
- **Part 3 (Advanced concepts):** type system, typing rules, type checking, soundness/completeness, type inference, polymorphism, modules, module procedures, typed modules, objects, classes, methods, inheritance, typed object-oriented languages

Overview

We learn the language concepts by defining and implementing small languages:



Defining a Programming Language

We need to specify syntax and semantics of the language:

- Syntax: how to write programs
- Semantics: the meaning of the programs

Both are formally specified by inductive definitions and implemented in OCaml.

Let $\subseteq$ Expression

Syntax

$$\begin{array}{rcl} P & \rightarrow & E \\ E & \rightarrow & n \\ & | & x \\ & | & E + E \\ & | & E - E \\ & | & \texttt{zero?}\ E \\ & | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\ & | & \texttt{let}\ x = E\ \texttt{in}\ E \end{array}$$

Semantics

How can we express the meaning of `while` loop?

```
whlie B do C
```

- Informal semantics: *"The command C is executed repeatedly so long as the value of the expression B remains true. The test takes place before each execution of the command"*.
  - intuitive and suitable for humans
  - ambiguous and not suitable for rigorous reasoning
- Formal semantics: The meaning is defined in mathematics:

$$\frac{M \vdash E \Rightarrow \mathit{false}}{M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow \mathit{true} \quad M \vdash C \Rightarrow M_1 \quad M_1 \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}{M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}$$

  - no confusion
  - a basis for reasoning about program behaviors

Values and Environments

To define the semantics, we define values and environments.

- The set of values that the language manipulates, e.g., in Let,

$$\mathbf{Val} = \mathbb{Z} + \mathbf{Bool}$$

- Environments maintains variable bindings:

$$\mathbf{Env} = \mathbf{Var} \to \mathbf{Val}$$

  Notations:
  - ρ ranges over environments, i.e., $\rho \in \mathbf{Env}$.
  - $[]$: the empty environment.
  - $[x \mapsto v]\rho$: the extension of ρ where x is bound to v:

$$([x \mapsto v]\rho)(y) = \begin{cases} v & \text{if } x = y \\ \rho(y) & \text{otherwise} \end{cases}$$

  - $[x_1 \mapsto v_1, x_2 \mapsto v_2]\rho$: the extension of ρ where x_1 is bound to v_1, x_2 to v_2:

$$[x_1 \mapsto v_1, x_2 \mapsto v_2]\rho = [x_1 \mapsto v_1]([x_2 \mapsto v_2]\rho)$$

Semantics

$\boxed{\rho \vdash e \Rightarrow v}$: the value of e in ρ is v.

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 - E_2 \Rightarrow n_1 - n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{zero?}\ E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{zero?}\ E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

A program e has semantics w.r.t. ρ iff we can derive $\rho \vdash e \Rightarrow v$ for some value v starting from the axioms and applying the inference rules finitely many times.

Example: Arithmetic Expressions

- In $\rho = [i \mapsto 1, v \mapsto 5, x \mapsto 10]$, program $(x - 3) - (v - i)$ has semantics and its value is 3, because

$$\dfrac{\dfrac{\overline{\rho \vdash x \Rightarrow 10} \quad \overline{\rho \vdash 3 \Rightarrow 3}}{\rho \vdash x - 3 \Rightarrow 7} \qquad \dfrac{\overline{\rho \vdash v \Rightarrow 5} \quad \overline{\rho \vdash i \Rightarrow 1}}{\rho \vdash v - i \Rightarrow 4}}{\rho \vdash (x - 3) - (v - i) \Rightarrow 3}$$

- But expression $y - 3$ does not have semantics because

$$\rho \vdash y - 3 \Rightarrow v$$

cannot be derived for any value v.

- In $\rho = [x \mapsto true]$, the semantics of $x + 1$ is not defined because

$$\rho \vdash x + 1 \Rightarrow v$$

cannot be derived for any v.

Example: Conditional Expression

In $\rho = [x \mapsto \mathbf{33}, y \mapsto \mathbf{22}]$,

$$\texttt{if zero? } (x - \mathbf{11}) \texttt{ then } y - \mathbf{2} \texttt{ else } y - \mathbf{4}$$

is well-defined and its value is $\mathbf{18}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\rho \vdash x \Rightarrow \mathbf{33}} \qquad \dfrac{}{\rho \vdash \mathbf{11} \Rightarrow \mathbf{11}}}{\rho \vdash x - \mathbf{11} \Rightarrow \mathbf{22}}}{\rho \vdash \texttt{zero? } (x - \mathbf{11}) \Rightarrow \mathit{false}} \qquad \dfrac{\dfrac{}{\rho \vdash y \Rightarrow \mathbf{22}} \qquad \dfrac{}{\rho \vdash \mathbf{4} \Rightarrow \mathbf{4}}}{\rho \vdash y - \mathbf{4} \Rightarrow \mathbf{18}}}{\rho \vdash \texttt{if zero? } (x - \mathbf{11}) \texttt{ then } y - \mathbf{2} \texttt{ else } y - \mathbf{4} \Rightarrow \mathbf{18}}$$

Example: Let Expression

A `let` expression creates a new *variable binding* in the environment: e.g.,

- $$\dfrac{[] \vdash \mathbf{5} \Rightarrow \mathbf{5} \qquad \dfrac{\dfrac{}{[x \mapsto \mathbf{5}] \vdash x \Rightarrow \mathbf{5}} \qquad \dfrac{}{[x \mapsto \mathbf{5}] \vdash \mathbf{3} \Rightarrow \mathbf{3}}}{[x \mapsto \mathbf{5}] \vdash x - \mathbf{3} \Rightarrow \mathbf{2}}}{[] \vdash \texttt{let}\ x = \mathbf{5}\ \texttt{in}\ x - \mathbf{3} \Rightarrow \mathbf{2}}$$

- In $[x \mapsto \mathbf{7}, y \mapsto \mathbf{2}]$, the program

$$\texttt{let}\ y = (\texttt{let}\ x = x - \mathbf{1}\ \texttt{in}\ x - y)\ \texttt{in}\ x - \mathbf{8} - y$$

evaluates to $-\mathbf{5}$:

$$\dfrac{\dfrac{\dfrac{\cdots}{[x \mapsto \mathbf{7}, y \mapsto \mathbf{2}] \vdash x - \mathbf{1} \Rightarrow \mathbf{6}} \qquad \dfrac{\cdots}{[x \mapsto \mathbf{6}, y \mapsto \mathbf{2}] \vdash x - y \Rightarrow \mathbf{4}}}{[x \mapsto \mathbf{7}, y \mapsto \mathbf{2}] \vdash \texttt{let}\ x = x - \mathbf{1}\ \texttt{in}\ x - y \Rightarrow \mathbf{4}} \qquad \dfrac{.}{[x \mapsto \mathbf{7}, y \mapsto \mathbf{4}] \vdash}}{[x \mapsto \mathbf{7}, y \mapsto \mathbf{2}] \vdash \texttt{let}\ y = (\texttt{let}\ x = x - \mathbf{1}\ \texttt{in}\ x - y)\ \texttt{in}\ x - \mathbf{8} - y \Rightarrow}$$

Implementation: Syntax

Syntax in OCaml:

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
and var = string
```

Examples:

```
# ADD (CONST 1, VAR "x");;
- : exp = ADD (CONST 1, VAR "x")
# IF (ISZERO (CONST 1), ADD (CONST 1, VAR "x"), CONST 3);;
- : exp = IF (ISZERO (CONST 1), ADD (CONST 1, VAR "x"), CONST 3)
```

Implementation: Values and Environments

Values:

```
type value = Int of int | Bool of bool
```

Environments:

```
type env = var -> value
let extend_env (x,v) e = fun y -> if x = y then v else (e y)
let apply_env e x = e x
```

Implementation: Semantics

```
let rec eval : exp -> env -> value
=fun exp env ->
  match exp with
   | CONST n -> Int n
   | VAR x -> apply_env env x
   | ADD (e1,e2) ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
       (match v1,v2 with
       | Int n1, Int n2 -> Int (n1 + n2)
       | _ -> raise (Failure "Type Error: non-numeric values"))
   | SUB (e1,e2) ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
       (match v1,v2 with
       | Int n1, Int n2 -> Int (n1 - n2)
       | _ -> raise (Failure "Type Error: non-numeric values"))
   ...
```

Code Reuse by Higher-Order Functions

The common pattern in ADD and SUB can be extracted by

```
let rec eval_bop: (int -> int -> int) -> exp -> exp -> env -> value
=fun op e1 e2 env ->
  let v1 = eval e1 env in
  let v2 = eval e2 env in
    (match v1,v2 with
    | Int n1, Int n2 -> Int (op n1 n2)
    | _ -> raise (Failure "Type Error: non-numeric values for +"))
```

With `eval_bop`,

```
  | ADD (e1,e2) -> eval_bop (+) e1 e2 env
  | SUB (e1,e2) -> eval_bop (-) e1 e2 env
```

Implementation: Semantics

```
let rec eval : exp -> env -> value
=fun exp env ->
  ...
  | ISZERO e ->
    (match eval e env with
    | Int n when n = 0 -> Bool true
    | _ -> Bool false)
  | IF (e1,e2,e3) ->
    (match eval e1 env with
    | Bool true -> eval e2 env
    | Bool false -> eval e3 env
    | _ -> raise (Failure "Type Error: condition must be Bool type")
  | LET (x,e1,e2) ->
    let v1 = eval e1 env in
      eval e2 (extend_env (x,v1) env)
```

Example

Running the program:

```
let run : program -> value
=fun pgm -> eval pgm empty_env
```

Examples:

```
# let e1 = LET ("x", CONST 1, ADD (VAR "x", CONST 2));;
val e1 : exp = LET ("x", CONST 1, ADD (VAR "x", CONST 2))
# run e1;;
- : value = Int 3
```

Summary

Designed and implemented Let:

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & \texttt{zero?}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E
\end{array}
$$

- how to formally specify syntax and semantics of programming languages
- key language concepts: environment and binding
- how to implement the language specification

Homework

- Download `let.ml`, the implementation of Let.
- Represent and Evaluate the following programs:
  - ```
    let x = 7
    in let y = 2
       in let  y = let x = x - 1
                   in x - y
          in (x-8)-y
    ```
  - ```
    let z = 5
    in let x = 3
       in let y = x - 1
          in let x = 4
               in z - (x-y)
    ```