CS 321 Programming Languages

Environments and Closures

Baris Aktemur

Özyeğin University

Last update made on Wednesday 9th November, 2016 at 08:17.

Some of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at `http://courses.engr.illinois.edu/cs421`

Environments

An **environment** is a set of bindings. It keeps record of what value is associated with a given name.

A key concept in programming language semantics and implementation.

Notation

$\rho = \{name_1 \mapsto v_1, name_2 \mapsto v_2, \ldots\}$
Note that an environment defines a partial function.

An environment is often implemented as a list or stack, or a stack of lists.

```
# let test = 3 < 2;;
val test : bool = false
(* ρ1 = {test ↦ false} *)
# let a = 1
  and b = a + 4;;
val a : int = 1
val b : int = 5
(* ρ2 = {test ↦ false, a ↦ 1, b ↦ 5} *)
# let a = 3;;
val a : int = 3
(* ρ3 = {test ↦ false, a ↦ 3, b ↦ 5} *)
(* New bindings hide old *)
```



```
(* ρ1 = {a ↦ 4} *)
# let c = 42;;
val c : int = 42
(* ρ2 = {c ↦ 42, a ↦ 4} *)

# let k = let c = a - 1
          (* ρ3 = {c ↦ 3, a ↦ 4} *)
          in c * a;;
val k : int = 12
(* ρ4 = {c ↦ 42, a ↦ 4, k ↦ 12} *)
# k;;
- : int = 12
# c;;
- : int = 42
```

Function values

- Functions are first-class values in OCaml.
- They can be passed as argument, returned from functions, bound to variables, etc.
- What value should we keep in the environment for a function?

Answer

A **closure**: a tuple of the function parameters, function body, and the environment in effect at the point the function is declared.

```
(* ρ1 = {...} *)
# let addFive x = x + 5;;
val add : int -> int = <fun>
(* ρ2 = {addFive ↦ ⟨x → x + 5, ρ1⟩, ...} *)
# addFive;;
- : (int -> int) = <fun>
```


Closures

```
(* ρ1 = {} *)
# let x = 17;;
(* ρ2 = {x ↦ 17} *)
# let addX y = x + y;;
(* ρ3 = {addX ↦ ⟨y → x + y, ρ2⟩, x ↦ 17} *)
# let x = 55;;
(* ρ4 = {addX ↦ ⟨y → x + y, ρ2⟩, x ↦ 55} *)
# addX 25;;
- : int = 42
```

Evaluation of function application with Static Scoping

Given an application expression $e_1 e_2$ in an environment ρ:

- Evaluate e_1 in ρ, obtain a closure $\langle x \to e_b, \rho_f \rangle$.
- Evaluate e_2 in ρ, obtain a value v.
- Bind v to x to extend ρ_f. That is, obtain $\rho_b = \{x \mapsto v\} + \rho_f$.
- Evaluate e_b in environment ρ_b.

Static scoping example

Evaluate `addx 25`, assuming the environment
$\rho_3 = \{addx \mapsto \langle y \to x + y, \rho_2 \rangle, x \mapsto 55\}$.

- Evaluate `addx` in ρ_3: gives $\langle y \to x + y, \rho_2 \rangle$.
- Evaluate 25 in ρ_3: trivially gives 25.
- Bind 25 to y to extend ρ_2: gives $\rho_b = \{y \mapsto 25\} + \{x \mapsto 17\}$.
- Evaluate $x + y$ in environment ρ_b: gives $25 + 17 = 42$.

Term

Note that we are evaluating the function using the environment that was saved in the closure (where x is 17); we are NOT using the current environment (where x is 55).
This is called **static scoping**.

Evaluation of function application with Dynamic Scoping

Given an application expression $e_1 e_2$ in an environment ρ:

- Evaluate e_1 in ρ to obtain a closure $\langle x \to e_b \rangle$. (Note: no environment saved!)
- Evaluate e_2 in ρ to obtain a value v.
- Bind v to x to extend ρ. That is, extend the current enviroment to obtain $\rho_b = \{x \mapsto v\} + \rho$.
- Evaluate e_b in environment ρ_b.

Dynamic scoping example

Evaluate `addx 25`, assuming the environment $\rho_3 = \{addx \mapsto \langle y \to x + y \rangle, x \mapsto 55\}$.

- Evaluate `addx` in ρ_3: gives $\langle y \to x + y \rangle$.
- Evaluate 25 in ρ_3: trivially gives 25.
- Bind 25 to y to extend ρ_3: gives $\rho_b = \{y \mapsto 25\} + \{addx \mapsto \langle y \to x + y \rangle, x \mapsto 55\}$.
- Evaluate $x + y$ in environment ρ_b: gives $25 + 55 = 80$.

Term

Note that we are evaluating the function using the current environment, which may be different each time function is applied. This is called **dynamic scoping**.

Static vs. Dynamic Scoping

- Dynamic scoping is easier to implement an interpreter/compiler. Lisp, Perl, Clojure have dynamic scoping.
- Static scoping is used in almost all the languages, because it is harder for the programmer to reason about a program (e.g. for debugging, for understanding a program, etc.) when using dynamic scoping.