Lecture 20 — Environment model

- The environment model is an alternative to the substitution model, which gives the same results but is more realistic.
- Today we will:
  - Look at more examples of higher-order functions
  - Discuss a different model of evaluation - the environment model
  - Discuss compilation of functional languages

map

- **The most famous of all higher-order functions:**

```
let rec map f lis = if lis=[] then []
                    else (f (hd lis)) :: map f (tl lis);;
```

  - `map (fun x->x+1) [1;2;3]`
  - `let incrBy n lis = map (fun x -> x+n) lis`
  - `let incrBy n = map (fun x -> x+n)`

- **Type of `map`?**

map exercises

- **addpairs: (int * int) list $\rightarrow$ int list**

- **appendString: string $\rightarrow$ string list $\rightarrow$ string list concatenates the first argument to the end of every string in the second argument**

- **incrall: int list list $\rightarrow$ int list list increments every element of every list in its argument**

fold_right

- **Usually called** `reduce`**, but called** `fold_right` **in OCaml:**

```
let rec fold_right (f:'a->'b->'b) (lis:'a list) (z:'b) : 'b
      = if lis=[] then z else f (hd lis) (fold_right f (tl lis) z)
```

- `fold_right (fun s s' -> s @ s') ["a"; "b"; "c"] ""`

- `fold_right (fun x y -> x+y) [3;4;5] 0`

- `fold_right (fun x y -> x::y) [3;4;5] []`

- `let h f lis = fold_right (fun x y -> (f x)::y) lis []`

Dictionaries as functions

- **Define a "dictionary" to be a function from strings to ints. Consider this definition of the basic operations:**

```
type dictionary = (string * int) list
let emptyDict = []
let rec lookup k d = if d=[] then -1
                     else if k = fst (hd d) then snd (hd d)
                                            else lookup k (tl d)
let add k v d = (k,v) :: d
```

Dictionaries as functions

- **Define the** *characteristic function* **of a dictionary** `d` **to be** `fun k -> lookup k d`**.**
- **What are the characteristic functions of these dictionaries:**
  - `emptyDict`
  - `add "a" 3 emptyDict`
  - `add "b" 4 (add "a" 3 emptyDict)`
  - `add "a" 5 (add "b" 4 (add "a" 3 emptyDict))`

Dictionaries as functions

- **Can represent dictionaries directly as characteristic functions:**

```
type dictionary = string -> int
let emptyDict = fun k -> -1
let rec lookup k d = d k
let add k v d = fun k' -> if k'=k then v else d k'
```

- `lookup "a" (add "a" 3 emptyDict)` $\Downarrow$

Dictionaries as functions

- `lookup "a" (add "b" 4 (add "a" 3 emptyDict))` $\Downarrow$

Dictionaries as functions (v. 2)

- **Returning -1 when a name is not in the dictionary is not such a good plan. Suppose** `lookup` **in the list representation above were redefined this way:**

```
let rec lookup k d = if d=[] then raise NotBoundException
                     else if k = fst (hd d) then snd (hd d)
                                            else lookup k (tl d)
```

- **Define** `emptyDict`**,** `lookup`**, and** `add` **in the characteristic function representation.**

Dictionaries as functions (v. 3)

- **Another approach to handling the unbound name issue is to use the "**`option`**" type in OCaml:**

```
type 'a option = Some of 'a | None
```

- `lookup` **in the list representation, using** `int option`**:**

```
let rec lookup k d = if d=[] then None
                     else if k = fst (hd d) then Some (snd (hd d))
                          else lookup k (tl d)
```

- **Define** `emptyDict`, `lookup`, **and** `add` **in the characteristic function representation.**

Evaluation in the environment model

- **Substitution model is easy to understand, but it does not reflect how actual implementations work.**
  - **To apply function `Fun(`x`,` e`)` to value v, instead of creating a new copy of e with all the x's replaced by v's, just *record* that x has value v in a separate data structure, called an *environment*.**
  - **All expression evaluation occurs "within" an environment.**
  - **To remember the values of variables in a function `fun` x `->` e, need to create a *closure* $<$ `fun` $x \rightarrow e, \rho >$.**

Environment model evaluation rules

(Const) Const c, $\rho \Downarrow$ Const c

(Var) a, $\rho \Downarrow \rho(a)$

(Fun) Fun(a,e), $\rho \Downarrow$ <Fun(a,e), ρ >

(Rec) Rec(f,e), $\rho \Downarrow$ < Rec(f,e), ρ >

(δ) $e\ op\ e'$, $\rho \Downarrow v\ OP\ v'$
e, $\rho \Downarrow v$
e', $\rho \Downarrow v'$

(δ) $op\ e$, $\rho \Downarrow OP\ v$
e, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ True
e_2, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ False
e_3, $\rho \Downarrow v$

(List) [e_1, . . ., e_n], $\rho \Downarrow$ [v_1, . . ., v_n]
e_1, $\rho \Downarrow v_1$
⋮
e_n, $\rho \Downarrow v_n$

(Let) Let(a,e,e'), $\rho \Downarrow v'$
e, $\rho \Downarrow v$
e', $\rho[a \mapsto v] \Downarrow v'$

(App) $e\ e'$, $\rho \Downarrow v$
e, $\rho \Downarrow$ <Fun(a, e''), ρ' >
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v'] \Downarrow v$

(App) $e\ e'$, $\rho \Downarrow v''$
e, $\rho \Downarrow v$,
where v = <Rec(f, Fun(a, e'')), ρ' >
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

Evaluation in environment model

- $\emptyset$ **denotes the empty environment. We may write** $\emptyset[x \mapsto v]$ **as** $\{x \mapsto v\}$**.**

`let x = 3 in x+1`, $\emptyset$

`(fun x -> x+1) 3`, $\emptyset$

Evaluation in environment model

`((fun x -> fun y -> x+y) 3) 4,` $\emptyset$

Evaluation in environment model

`let f = fun x -> fun y -> x+y,` $\emptyset$
`in let g = f 1 in g 2`

Compilation of MiniOCaml

- **Compilation of functional languages starts from environment model.**
- **Need to discuss:**
  - **Representation of environments and closures, and variable look-up**
  - **Run-time structures (stack and heap)**
  - **Compilation rules, esp. for application and abstraction**

Representation of environments

- **Suppose we represent environments by** `(string * value) list`**. App rule becomes:**

  (App) $e\ e'$**,** $\rho \Downarrow v$

  e**,** $\rho \Downarrow <$**Fun(**a**,** e''**),** $\rho' >$
  e'**,** $\rho \Downarrow v'$
  e''**,** $(a, v') :: \rho' \Downarrow v$

  and the variable rule does a recursive list look-up.

- **Crucial question: Given a variable reference, can we determine** *at compile time* **where in the list it will occur?**

Representation of environments (cont.)

- **For any variable reference, crucial number is *the number of declarations (let or fun) intervening between the reference and the variable's declaration*.**

- **We will assume that the type-checking phase of the compiler has marked every variable reference with this number. E.g.**

```
fun x -> let f = fun y -> x + (let y = y+1 in x+y))
         in f x
```

Representation of environments (cont.)

- **Represent environment by linked list of <u>values</u> — names are not needed.**
- **An expression is executed in a "current" environment. Suppose register %env points to the head of the current environment. Rule for variable reference:**

  (Var) $a_k \rightsquigarrow$ **[MOV %tmp,%env; LOADIND %tmp,4(%tmp);...(**$k\ times$**)...]**

- **Now, we just have to make sure current environment is correctly formed; happens in abstraction and application rules (and let and letrec).**

Runtime state

- For concreteness, assume this state layout (all subject to change for real implementation, of course):
  - Stack consists of frames having exactly two addresses:
    - Return address — pointer into code of calling function
    - Calling function's environment
  - Register %env points to head of current environment. Register %ret used for return values
  - Environments are linked lists of primitive values and pointers to "heap values" — lists, tuples, closures
  - Closures are heap-allocated pairs, containing pointer to code and pointer to environment

Compilation

- **Will give compilation rules only informally.**
- **Compilation rules correspond closely to rules of environment model.**
- **Compilation rules for abstraction and application are same for static and dynamic typing. (Only rules for primitive operations change.)**
- **Compilation rules for other stuff — built-in operations, if — are normal, e.g.**

(Var) $e1 + e2$, **loc** $\leadsto$ **il1 @ il2 @ [ADD loc,loc1,loc2]**
  $e1$, **loc1** $\leadsto$ **il1**
  $e2$, **loc2** $\leadsto$ **il2**

Compiling abstractions

- **An abstraction does not involve any real computation — just creates a closure. However, the body of the abstraction must be compiled a little differently from an ordinary expression; it has to include code for function return at the end.**

- **In `fun` x `->` e, e should be compiled like this, somewhere in memory:**

(Function body) e **as function body** $\rightsquigarrow$ **il @ [move loc into %ret,**
then return from function (restore env. pointer and get return address from stack frame; pop stack; jump to return address)]
e, **loc** $\rightsquigarrow$ **il**

- **Suppose this code is at location m_f. The abstraction itself is compiled like this:**

(Fun) fun x -> e, **loc** $\rightsquigarrow$ **[loc = allocate closure in heap; move m_f, %env into closure]**

Compiling applications

- **Argument must evaluate to a (pointer to a) closure; this is where environments are built.**

(App) $e1\ e2$, **loc** $\rightsquigarrow$ **il1 @ il2 @ [function pointer** m_f **= loc1[0], environment pointer** ep **= loc1[1]; create new environment** ep' **by cons'ing loc2 to** ep**; push new stack frame, storing** m_{ret} **and %env; %env =** ep'**; JUMP** m_f**]**

$e1$, **loc1** $\rightsquigarrow$ **il1**

$e2$, **loc2** $\rightsquigarrow$ **il2**

where m_{ret} **is address of the next instruction after this code.**

Wrap-up

- **Today we discussed higher-order functions, the environment model of evaluation, and compilation of functional languages.**
- **We did this to get a better idea of how functional languages are implemented, and how to use them.**
- **What to do now:**
  - **MP10**