Lecture 21 — More higher-order functions

- In preparation for MP 11, we will look at more uses of higher-order functions, especially for "combinator-style programming."
- Today we will:
  - Finish discussion of compilation of functional languages
  - Discuss using higher-order functions to write parsers
  - Discuss MP 11

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) $[e_1, \ldots, e_n]$, $\rho \Downarrow [v_1, \ldots, v_n]$
- e_1, $\rho \Downarrow v_1$
- $\vdots$
- 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'', $\rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

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:**

$$\textbf{(App)}\ e\ e',\ \rho \Downarrow v$$
$$e,\ \rho \Downarrow <\textbf{Fun}(a,\ e''),\ \rho' >$$
$$e',\ \rho \Downarrow v'$$
$$e'',\ (a, v') :: \rho' \Downarrow v$$

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) $a_k \leadsto$ **[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 $\leadsto$ 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 $\leadsto$ il

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

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

Compiling applications

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

(App) $e1\ e2$, **loc** $\leadsto$ **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** $\leadsto$ **il1**

$e2$, **loc2** $\leadsto$ **il2**

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

Combinator-style programming

- **Can write complex programs by defining a library of higher-order functions and applying them to one another (and to built-in functions). These functions are called** *combinators* **(technically just a term for a closed expression).**
- **Makes it easy to create functions within the domain for which the combinators is designed.**
- **Is associated with notion of "domain-specific languages," especially when the functions are defines as infix operators; result "looks like" a language for the specific domain.**

Parser combinators

- **A** ***parser*** **is a function of type** `token list` $\rightarrow$ `(token list) option`**. (Recall** `type` α `option = Some` α $|$ `None`**.)**
- **Idea is to define functions that build parsers, rather than building parsers "by hand."**
- **E.g. Parser to recognize a single token:**

```
let token s = fun cl -> if cl=[] then None
                        else if s=hd cl then Some (tl cl)
                        else None;;
let parsex = token x;;
parsex [x];;
parsex[a];;
```

Parser combinators (cont.)

- **Parser combinators combine parsers to make more complicated parsers:**

```
let (++) p q = fun cl -> match p cl with None -> None
                                         | Some cl' -> q cl';;

let parsexy = token x ++ token y
parsexy [x, y]
parsexy [x, z]
```

- **(Note use of infix operator.)**

Parser combinators (cont.)

```
let (||) p q = fun cl ->  match p cl with None -> q cl
                                       | Some cl' -> Some cl';;
let parsexyorz = parsexy || token z
parsexyorz[x, y]
parsexyorz [z]
```

Parser combinators (cont.)

- **Put this together to define parser for grammar:**

```
A  ->  aB  | b
B  ->  cB  |  A
```

```
let rec parseA cl = ((token 'a' ++ parseB) || token 'b') cl
    and parseB cl = ((token 'c' ++ parseB) || parseA) cl;;

parseA ['a';'c';'c';'a';'b]
```

Why parser combinators?

- **Advantages of this approach:**
  - **Convenience:**

```
let epsilon : parser = fun toklis -> toklis
let rec kstar (p:parser) : parser
        = fun toklis -> (p ++ (kstar p)) toklis || epsilon
alist = kstar (token 'a')
```

  - **Can write entirely different parsers:**

```
let handle_includes (p:parser) : parser
   = fun toklis ->
      if hd toklis = "#include"
        then let new_toks
                   = lex (readfile (hd (tl toklis)))
        in p (new_toks @ tl (tl toklis)
        else p toklis
```

MP 11

- **You will get practice with higher-order functions by filling in code in a combinator-based picture-drawing system.**
- **In this combinator library, the following definitions are key:**

```
type point = float * float

type transformation = point -> point

type draw_cmd = Pixel of point
              | Line of point * point
              | Oval of point * point * point

type picture = transformation -> draw_cmd list
```

Wrap-up

- **Today we discussed a specific example of the use of higher-order functions, a "combinator library" for parsing.**
- **We did this for more practice with higher-order functions.**
- **What to do now:**
  - **MP11**