Schedule

SCHEDULE

w	date	topic
~~7~~	~~November 25~~	~~introduction~~
~~8~~	~~December 2~~	~~higher-order functions, lists, trees~~
~~9~~	~~December 9~~	~~graphs, combinatorics~~
~~10~~	~~December 16~~	~~program reasoning~~
11	January 13	λ and interpreter
12	January 20	type system
13	January 27	exam part 2

CONTENTS

1. eval
2. λ, β, δ, core ML
3. let rec
4. eval

Evaluator

Evaluator for Arithmetic

$$e ::= c \mid x \mid e + e \mid e - e$$

```
type e =                              (* expression *)
  | Const of int
  | Var of string
  | Add of e * e
  | Sub of e * e

type value = int                      (* semantical value *)
type env = (string * value) list      (* environment *)
exception Unbound of string
```

EXERCISE

```
# lookup [("x", 10); ("y",2)] "y"
- : int = 2
# eval [("x",10)] (Sub (Var "x", Const 1));;
- : int = 9
```



```
let lookup env x =
let rec eval env =
  | Const n -> n
  | Var x -> lookup env x
  | Add (e1, e2) ->
  | Sub (e1, e2) ->
```

λ, β, δ, core ML

λ-calculus

$e ::=$ ~~c~~	~~constant~~	$\mid$ ~~$e + e$~~
$\mid x$	variable	$\mid$ ~~$e - e$~~
$\mid e\ e$	application	$\mid$ ~~List.map e e~~
$\mid$ fun $x \to e$	λ abstraction	$\mid$ ~~List.filter e e~~
$\mid$ ~~let $x = e$ in e~~	~~definition~~	$\mid$ ~~$\cdots$~~
$\mid$ ~~let rec $x = e$ in e~~	~~recursive def.~~	
$\mid$ ~~if e then e else e~~	~~if expression~~	

too messy... which part is essential for computation?

core ML

$e ::= x \mid e\ e \mid$ fun $x \to e$ λ-calculus

$\mid c$ for primitives

$\mid$ let $x \to e$ for polymorphism

$$e ::= c \mid x \mid e\ e \mid \text{fun } x \to e$$
$$v ::= c \mid \text{fun } x \to e$$

- OCaml program is expression without free-variables
- call by value strategy

EXERCISE

call by value strategy

- $(0 + (2 * 2)) - (3 + 4)$
- $(\text{fun } x \to 1)\ (1 + (4 * 3))$

Reductions

in call by value strategy distinguish two reductions

- β reduces $(\text{fun } x \to e)\ v$ (function application)
- δ reduces $c\ v\ \cdots\ v$ (primitive operation)

EXAMPLE

$$\begin{aligned} & (\text{fun } x \to (\text{fun } y \to +\ x\ y))\ 1\ 2 \\ \to_\beta\ & (\text{fun } y \to +\ 1\ y)\ 2 \\ \to_\beta\ & +\ 1\ 2 \\ \to_\delta\ & 3 \end{aligned}$$

REMARK

$$\text{fun } x \to +\ 0\ 1$$

is irreducible in call by value strategy

Define Missing Parts

Fixed Point Combinators

Fixed Point Combinators

PROBLEM

core ML does not contain let rec. how to implement recursion?

SOLUTION call-by-value Y-combinator

let rec $f = e_1$ in e_2 is equivalent to let $f =$ fix (fun $f \rightarrow e_1$) in e_2

```
$ ocaml
# let fix =
    fun f -> (fun x -> f (fun y -> x x y))
             (fun x -> f (fun y -> x x y));;
This expression has type 'a -> 'b -> 'c
but is here used with type 'a
```


Fixed Point Combinator

```
$ ocaml -rectypes
# let fix =
    fun f -> (fun x -> f (fun y -> x x y))
             (fun x -> f (fun y -> x x y));;
val fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

EXERCISE

do not use let rec but use fix to implement sum

```
# let rec sum =
    fun sum n -> if n = 0 then 0 else n + sum (n - 1) in
  sum 10;;

# let sum =
    fix (fun sum n -> if n = 0 then 0 else n + sum (n - 1))
in
  sum 10;;
- : int = 55
```

Evaluator

Values, Environments and Closures

GOAL

eval env e returns value v such that $e \rightarrow^* v$ in call by value strategy under env

```
let rec eval env = function
  | Const c         ->
  | Var x           -> lookup env x
  | Fun (x, e)      ->
  | App (e1, e2)    ->
  | Let (x, e1, e2) ->
```


Environments

```
type env = (string * value) list

let rec eval env = function
  | Const c         ->
  | Var x           -> lookup env x
  | Fun (x, e)      ->
  | App (e1, e2)    ->
  | Let (x, e1, e2) -> eval (extend env x (eval env e1)) e2
```

PROBLEM

what is value?

$$\text{eval } [\,]\ (\text{Const } (\text{Int } 1)) = 1?$$
$$\text{eval } [\,]\ (\text{Const } (\text{Prim } (\text{"+"}, 2))) = ???$$
$$\text{eval } [(x, 1)]\ (\text{Fun } (y, +\ x\ y)) = ???$$

SOLUTION

introduce closures as well as values for constants

Values

```
type value = Constant of const * value list
           | Closure of string * expr * env
and env = (string * value) list

let rec eval env = function
  | Const c         -> Constant (c,[])
  | Var x           -> lookup env x
  | Fun (x, e)      -> Closure (x, e, env)
  | App (e1, e2)    ->
  | Let (x, e1, e2) -> eval (extend env x (eval env e1)) e2
```

PROBLEM

how to apply?

Apply

```
type value = Constant of const * value list
           | Closure of string * expr * env
and env = (string * value) list

let rec eval env = function
  | Const c         -> Constant (c,[])
  | Var x           -> lookup env x
  | Fun (x, e)      -> Closure (x, e, env)
  | App (e1, e2)    -> apply (eval env e1) (eval env e2)
  | Let (x, e1, e2) -> eval (extend env x (eval env e1)) e2
and apply v1 v2 =
```

PROBLEM

how to apply?

Apply

$$v_1 \circ v_2 = \text{apply } v_1 \ v_2$$

$$\begin{aligned}
& \text{eval } [\,] \ ((\text{fun } x \to \text{fun } y \to + \ x \ y) \ 1 \ 2) \\
= \ & \text{eval } [\,] \ ((\text{fun } x \to \text{fun } y \to + \ x \ y) \ 1) \ \circ \ \text{eval } [\,] \ 2 \\
= \ & (\text{eval } [\,] \ (\text{fun } x \to \text{fun } y \to + \ x \ y) \ \circ \ (\text{eval } [\,] \ 1)) \ \circ \ 2 \\
= \ & (\text{Closure}(x, \text{fun } y \to + \ x \ y, [\,]) \ \circ \ 1) \ \circ \ 2 \\
= \ & \text{eval } [(x,1)] \ (\text{fun } y \to + \ x \ y) \ \circ \ 2 \\
= \ & \text{eval } [(y,2);(x,1)] \ (+ \ x \ y) \\
= \ & \text{eval } [(y,2);(x,1)] \ (+ \ x) \ \circ \ \text{eval } [(y,2);(x,1)] \ y) \\
= \ & (\text{eval } [(y,2);(x,1)] \ (+) \ \circ \ \text{eval } [(y,2);(x,1)] \ x) \ \circ \ 2 \\
= \ & (\text{Constant}(+, [\,]) \ \circ \ 1) \ \circ \ 2 \\
= \ & \text{Constant}(+, [1]) \ \circ \ 2 \\
= \ & \delta(+, [1;2]) \\
= \ & 3
\end{aligned}$$

 — a lot of constructors are omitted here