Lecture 19 — Higher-order functions

- Functional languages to be "created" at run time (by filling in values of free variables). This capability is very powerful.
- Today we will look at examples of higher-order functions:
  - Simple examples
  - Currying and uncurrying
  - `map` and `fold_right`
  - Representing dictionaries as functions
- (There are a lot more examples in this lecture than we will get to in class; use the others to study for the midterm.)

Substitution model evaluation rules

(Const) Const x $\Downarrow$ Const x

(Fun) Fun(a,e) $\Downarrow$ Fun(a,e)

(Rec) Rec(f,Fun(a,e)) $\Downarrow$ Fun(a,e[Rec(f,Fun(a,e))/f]

(δ) $e\ op\ e' \Downarrow v\ OP\ v'$

$$e \Downarrow v$$
$$e' \Downarrow v'$$

(δ) $op\ e \Downarrow OP\ v$

$$e \Downarrow v$$

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

$$e_1 \Downarrow \text{True}$$
$$e_2 \Downarrow v$$

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

$$e_1 \Downarrow \text{False}$$
$$e_3 \Downarrow v$$

(List) $[e_1, \ldots, e_n] \Downarrow [v_1, \ldots, v_n]$

$$e_1 \Downarrow v_1$$
$$\vdots$$
$$e_n \Downarrow v_n$$

(App) $e\ e' \Downarrow v$

$$e \Downarrow \text{Fun}(a,\ e'')$$
$$e' \Downarrow v'$$
$$e''[v'/a] \Downarrow v$$

(Let) Let(a,e,e') $\Downarrow v'$

$$e \Downarrow v$$
$$e'[v/a] \Downarrow v'$$

Simple examples

- `add = fun x -> fun y -> x+y`   **(Type:** `int -> int -> int`**)**

`(add 3) 4` $\Downarrow$ `7`

    `add 3` $\Downarrow$ `fun y -> 3+y`

        `add` $\Downarrow$ `fun x -> fun y -> x+y`

        `3` $\Downarrow$ `3`

        `fun y -> 3+y` $\Downarrow$ `fun y -> 3+y`

    `4` $\Downarrow$ `4`

    `3+4` $\Downarrow$ `7`

Simple examples (cont.)

- `apply_to_10 = fun f -> f 10` **(Type:** `(int -> 'a) -> 'a`**)**

```
(apply_to_10 add) 4 ⇓ 14
    apply_to_10 add ⇓ fun y -> 10+y
        apply_to_10 ⇓ fun f -> f 10
        add ⇓ fun x -> fun y -> x+y
        (fun x -> fun y -> x+y) 10 ⇓ fun y -> 10+y
            (fun x -> fun y -> x+y) ⇓ (fun x -> fun y -> x+y)
            10 ⇓ 10
            (fun y -> 10+y) ⇓ (fun y -> 10+y)
    4 ⇓ 4
    10+4 ⇓ 14
        10 ⇓ 10
        4 ⇓ 4
```

Simple examples (cont.)

- `double = fun f -> fun x -> f (f x)` **(Type:** `('a -> 'a) -> ('a -> 'a)`**)**
  `add6 = double (add 3)`

  add6 5 $\Downarrow$

```
add6 5 ⇓ 11
    double (add 3) ⇓ fun x -> (fun y -> 3+y)((fun y -> 3+y) x)
        double ⇓ fun f -> fun x -> f (f x)
        add 3 ⇓ fun y -> 3+y
            (fun x -> fun y -> x+y) ⇓ (fun x -> fun y -> x+y)
            3 ⇓ 3
            (fun y -> 3+y) ⇓ (fun y -> 3+y)
        fun x -> (fun y -> 3+y)((fun y -> 3+y) x) ⇓ fun x -> (fun y -> 3+y)((fun y -> 3+y) x)
    5 ⇓ 5
    (fun y -> 3+y)((fun y -> 3+y) 5) ⇓ 11
        fun y -> 3+y ⇓ fun y -> 3+y
        (fun y -> 3+y) 5 ⇓ 8
            fun y -> 3+y ⇓ fun y -> 3+y
            5 ⇓ 5
            3+5 ⇓ 8
                3 ⇓ 3
                5 ⇓ 5
        3+8 ⇓ 11
            3 ⇓ 3
            8 ⇓ 8
```

Simple examples (cont.)

- `compose = fun f -> fun g -> fun x -> f (g x)`

  (Type: `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`**)**

```
double = fun f -> compose f f
add6 = double (add 3)
```

```
add6 5 ⇓ 11
     double (add 3) ⇓ fun x -> (fun y -> 3+y)((fun y -> 3+y) x)
          double ⇓ fun f -> compose f f
          add 3 ⇓ fun y -> 3+y    (sub-evals omitted)
          (compose (fun y -> y+3)) (fun y -> y+3) ⇓ fun x -> (fun y -> y+3)((fun y -> y+3) x)
               compose (fun y -> y+3) ⇓ (fun g -> fun x -> (fun y -> y+3) (g x))
                    compose ⇓ fun f -> fun g -> fun x -> f (g x)
                    fun y -> y+3 ⇓ fun y -> y+3
                    (fun g -> fun x -> (fun y -> y+3) (g x))
                              ⇓ (fun g -> fun x -> (fun y -> y+3) (g x))
               fun y -> y+3 ⇓ fun y -> y+3
               fun x -> (fun y -> y+3)((fun y -> y+3) x)
                              ⇓ fun x -> (fun y -> y+3)((fun y -> y+3) x)
     5 ⇓ 5
     (fun y -> y+3)((fun y -> y+3) 5) ⇓ 11    (sub-evals omitted)
```

Currying

- **Functions of type $\tau \to \tau' \to \tau''$ (curried) and $\tau * \tau' \to \tau''$ (uncurried) *cannot be used interchangeably*:**

```
add = fun x -> fun y -> x+y;;  (Type:int -> int -> int)
let add_unc = fun p -> fst p + snd p;;  (Type:int * int -> int)
let use_curried = fun g -> g 3 4;;  (Type: (int -> int -> int) -> 'a)
let use_uncurried = fun g -> g(3,4);;  (Type: (int * int -> int) -> 'a)

use_curried add;;  (* 7 *)

use_uncurried add;;  (* type error *)

use_curried add_unc;;  (* type error *)

use_uncurried add_unc;;  (* 7 *)
```

Currying (cont.)

- **Can define functions `curry` and `uncurry`:**

```
let add_unc = fun p -> fst p + snd p;;
let use_curried = fun g -> g 3 4;;
let curry = fun f -> fun x -> fun y -> f(x,y)
```

```
use_curried (curry add_unc) ⇓ 7
     use_curried ⇓ fun g -> g 3 4
     curry add_unc ⇓ fun x -> fun y -> (fun p -> (fst p)+(snd p))(x,y)
          curry ⇓ fun f -> fun x -> fun y -> f(x,y)
          add_unc ⇓ fun p -> (fst p)+(snd p)
          fun x -> fun y -> (fun p -> (fst p)+(snd p))(x,y)
                             ⇓ fun x -> fun y -> (fun p -> (fst p)+(snd p))(x,y)
     (fun x -> fun y -> (fun p -> (fst p)+(snd p))(x,y)) 3 4 ⇓ 7
          (fun x -> fun y -> (fun p -> (fst p)+(snd p))(x,y)) 3
                             ⇓ fun y -> (fun p -> (fst p)+(snd p))(3,y)   (sub-evals omitted)
          4 ⇓ 4
          (fun p -> (fst p)+(snd p))(3,4) ⇓ 7
               (fun p -> (fst p)+(snd p)) ⇓ (fun p -> (fst p)+(snd p))
               (3,4) ⇓ (3,4)    (sub-eval's omitted)
               (fst (3,4))+(snd (3,4)) ⇓ 7    (sub-eval's omitted)
```

Type of `curry`: `('a * 'b -> 'c) -> 'a -> 'b -> 'c`

Currying (cont.)

```
let add x y = x + y;;
let use_uncurried g = g(3,4);;
let uncurry =  (* Type: ('a -> 'b -> 'c) -> 'a*'b -> 'c *)

    fun f -> fun p -> f (fst p) (snd p)

use_uncurried (uncurry add);;  (* 7 *)
```

Doing this *informally*:

uncurry add $\Downarrow$ fun p -> add (fst p) (snd p)

```
use_uncurried (uncurry add)
     = (uncurry add) (3, 4)
     = (fun p -> add (fst p) (snd p)) (3, 4)
     = add (fst (3,4)) (snd (3,4))
     = 7
```

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]     (* [2;3;4] *)`
  - `let incrBy n lis = map (fun x -> x+n) lis`

    Increment every element of lis by n.

  - `let incrBy n = map (fun x -> x+n)`

    Increment every element of lis by n.

- **Type of `map`?**

```
('a -> 'b) -> 'a list -> 'b list
```

map exercises

- **addpairs: (int * int) list → int list**

```
map (fun (x,y) -> x+y)
```

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

```
fun s -> map (fun s’ -> s’ ^ s)
```

- **incrall: int list list → int list list increments every element of every list in its argument**

```
map (map (fun x -> x+1))
```

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"] ""`

  `"abc"`

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

  `12`

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

  `[3;4;5]`

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

  `h` **is the same as** `map`**.**

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`

    `fun k -> lookup k emptyDict` = `fun k -> -1`
  - `add "a" 3 emptyDict`

    `fun k -> if k="a" then 3 else -1`
  - `add "b" 4 (add "a" 3 emptyDict)`

    `fun k -> if k="b" then 4 else if k="a" then 3 else -1`
  - `add "a" 5 (add "b" 4 (add "a" 3 emptyDict))`

    `fun k -> if k="a" then 5 else if k="b" then 4 else -1`

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)` $\Downarrow$

`lookup "a" (add "a" 3 emptyDict)` $\Downarrow$ 3
    `lookup "a"` $\Downarrow$ `fun d -> d "a"` (*sub-evals omitted*)
    `add "a" 3 emptyDict` $\Downarrow$ `fun k' -> if k'="a" then 3 else emptyDict "a"` (*sub-evals omitted*)
    `(fun k' -> if k'="a" then 3 else emptyDict "a") "a"` $\Downarrow$ 3 (*first two sub-evals omitted*)
        `if "a"="a" then 3 else emptyDict "a"` $\Downarrow$ 3
            `"a"="a"` $\Downarrow$ `true`
            3 $\Downarrow$ 3

Dictionaries as functions

- **We're starting to omit steps without comment, particularly the ones of the form $e \Downarrow e$.**

```
lookup "a" (add "b" 4 (add "a" 3 emptyDict)) ⇓ 3
   lookup "a" ⇓ fun d -> d "a"
   add "b" 4 (add "a" 3 emptyDict)
            ⇓ fun k' -> if k'="b" then 4 else (fun k' -> if k'="a" then 3 else emptyDict k') k'
      add "a" 3 emptyDict ⇓ fun k' -> if k'="a" then 3 else emptyDict k'
   (fun k' -> if k'="b" then 4 else (fun k' -> if k'="a" then 3 else emptyDict k') k') "a" ⇓ 3
      if "a"="b" then 4 else (fun k' -> if k'="a" then 3 else emptyDict k') "a" ⇓ 3
         "a"="b" ⇓ false
         (fun k' -> if k'="a" then 3 else emptyDict k') "a" ⇓ 3    (sub-evals omitted)
```

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.**

```
emptyDict = fun k -> raise NotBoundException
lookup, add -  unchanged
```

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.**

```
emptyDict = fun k -> None
add k v d = fun k' -> if k'=k then Some v else d k'
lookup - unchanged
```

More h-o function examples

- **Define the following functions, and give their types:**
  - `reverse_args` **takes a curried function of two arguments and returns the function that takes its arguments in the opposite order. E.g.**

```
let sub x y = x-y;;
(reverse_args sub) 4 3;;  (* returns -1 *)

let reverse_args = fun f -> fun x -> fun y -> f y x
```

```
Type: ('a -> 'b -> 'c) -> ('b -> 'a -> 'c)
```

More h-o function examples

- `fix_snd` **takes an *uncurried* function, and a value for its *second* argument, and returns a function of the first argument. Give the type of** `fix_snd`**, and define it using** `curry` **and** `reverse_args`**.**

```
let div (x, y) = x/y;;
let halve = fix_snd div 2;;
halve 10;;  (* returns 5 *)

let fix_snd f v = reverse_args (curry f) v
```

```
Type: ('a * 'b -> 'c) -> 'b -> ('a -> 'c)
```

Representing sets as functions

- **Implementing a "set of int" data type means defining a representation "`type intset =` *something*", and operations like:**
  - `let emptyset =` *something*
  - `let member (i:int) (s:intset) : bool =` *something*
  - `let add (i:int) (s:intset) : intset =` *something*

  The implementation is correct if it behaves in a "set-like" way, e.g. `member 3 emptyset` **= false,** `member 3 (add 4 (add 3 emptyset))` **= true, etc.**

- **In functional languages, a set can be represented by its** ***characteristic function*****:**

```
type intset = int -> bool
```

Representing sets as functions

```
type intset = int -> bool

let emptyset : intset =

          fun n -> false


let member (n:int) (s:intset) : bool =

          s n


let add (n:int) (s:intset) : intset =

         fun n' -> n'=n or s n'
```

Representing sets as functions (cont.)

```
let union (s1:intset) (s2:intset) : intset =

       fun n -> s1 n or s2 n

let intersection (s1:intset) (s2:intset) : intset =

       fun n -> s1 n && s2 n

let remove (n:int) (s:intset) : intset =

       fun m -> s m && m <> n

let complement (s:intset) : intset =

       fun n -> not (s n)

let intsAbove (n:int) : intset =

       fun m -> m > n
```

Wrap-up

- **Today we discussed higher-order functions, by going through examples.**
- **We did this because some of these functions are useful, and because they will help you understand more complicated uses of higher-order functions.**
- **What to do now:**
  - **MP9**
  - **Use examples in this lecture to study for exam**
  - **Review session: Sunday, 7pm, 1404 SC**