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) $\Downarrow v$

$e_1 \Downarrow$ True

$e_2 \Downarrow v$

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

$e_1 \Downarrow$ 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) $e\ e' \Downarrow v$

$e \Downarrow$ 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:)**

  `(add 3) 4` $\Downarrow$

Simple examples (cont.)

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

  `(apply_to_10 add) 4` $\Downarrow$

Simple examples (cont.)

- `double = fun f -> fun x -> f (f x)` **(Type:)**

  `add6 = double (add 3)`

  `add6 5` $\Downarrow$

Simple examples (cont.)

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

  (Type:)

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

  add6 $\Downarrow$

Simple examples (cont.)

- `add6 5` $\Downarrow$

Currying

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

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

use_curried add;;

use_uncurried add;;

use_curried add_unc;;

use_uncurried add_unc;;
```

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);;
```

Type of `curry`:

Currying (cont.)

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



use_uncurried (uncurry add);;
```

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.**

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 =




Type:
```

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;;  (* returns -1 *)
halve 10;;  (* returns 5 *)

let fix_snd f v =
```

`Type:`

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 =




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




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

Representing sets as functions (cont.)

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

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

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

```
let complement (s:intset) : intset =
```

```
let intsAbove (n:int) : intset =
```

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**