

Lecture 20 — Higher-order functions

- **Examples of higher-order functions**
 - `map`
 - `fold_right`
 - **Currying and uncurrying**
- **Representing sets as functions**
- **Representing environments as functions**

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: $(\text{int} * \text{int}) \text{ list} \rightarrow \text{int list}$**
- **appendString: $\text{string} \rightarrow \text{string list} \rightarrow \text{string list}$ concatenates the first argument to the end of every string in the second argument**
- **incrall: $\text{int list list} \rightarrow \text{int list list}$ 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 []`

Currying

- Functions in OCaml are automatically curried. That is,

```
let fun f x y z = e;;
```

is transformed to

```
let f = fun x -> fun y -> fun z -> e;;
```

which means it can be applied *partially*:

```
let g = f 5;;
```

$g\ 6\ 7 \iff \text{the same as } f\ 5\ 6\ 7.$

- Note difference between `fun x -> fun y -> e` and `fun (x,y) -> e`.

Currying

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

```
let f_curried x y = x + y;;  
let f_uncurried (x,y) = x + y;;  
let usecurried g = g 3 4;;  
let useuncurried g = g(3,4);;
```

```
usecurried f_curried;;
```

```
useuncurried f_curried;;
```

```
usecurried f_uncurried;;
```

```
useuncurried f_uncurried;;
```

Currying (cont.)

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

```
let curry (f:'a * 'b -> 'c) : ('a -> 'b -> 'c) =
```

```
let uncurry (f:'a -> 'b -> 'c) : ('a * 'b -> 'c) =
```

```
usecurried (curry f_uncurried);;  
useuncurried (uncurry f_curried);;
```

More h-o function examples

- `reverse (f: 'a -> 'b -> 'c) : ('b -> 'a -> 'c)`

`reverse (fun x y -> x-y) 3 5 = 2`

- `twice (f: 'a -> 'a ) : ('a -> 'a)`

`twice (fun x -> x+1) 3 = 5`

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 =
```

Representing dictionaries as functions

- Similarly, implementing an *environment* (a.k.a. *dictionary*), as in our mps, means defining a representation “`type environment = something`”, and operations:
 - `let emptyEnv : environment`
 - `fetch (id:id) (env:environment) : value`
 - `extend (id:id) (v:value) (env:environment) : environment`
- These operations need to act in an “environment-like” way, e.g.
 - `fetch "x" emptyEnv` **throws an exception**
 - `fetch "x" (extend "x" (IntConst 3) emptyEnv)` **returns** `IntConst 3`