objective caml

Daniel Jackson
MIT Lab for Computer Science
6898: Advanced Topics in Software Design
March 18, 2002

topics for today

familiar notions (from Scheme)

- let bindings, functions, closures, lists

new notions (from ML)

- inferred types and parametric polymorphism
- side-effects and the unit type
- datatypes (variants)

functions

applying an anonymous function

```
#  (fun x -> 2 * x) 3;;
- : int = 6
```

declaring a function and applying it

```
# let dbl = fun x -> 2 * x;;
val dbl : int -> int = <fun>
# dbl 3;;
- : int = 6
```

functionals, or higher-order functions

```
# let twice = fun f -> (fun x -> (f (f x)));;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# (twice dbl) 3;;
- : int = 12
```

let bindings

a let expression binds a variable to a value

```
# let x = 3 and y = 4 in x + y;;
- : int = 7
```

read-eval-print-loop uses let instead of define

```
# let x = 5;;
val x : int = 5
# x;;
- : int = 5
```

recursive let

```
# let rec fact i = if i = 0 then 1 else i * fact (i - 1);;
val fact : int -> int = <fun>
# fact 4;;
- : int = 24
```

let vs. define

```
# let k = 5;;
val k : int = 5
# let f = fun x -> x + k;;
val f : int -> int = <fun>
# f 3;;
- : int = 8
# let k = 6;;
val k : int = 6
# f 3;;
- : int = 8
```

let is lexical

- no side-effecting top-level define built-in

tuples

tuple constructor

```
# let x = 1, 2;;
val x : int * int = 1, 2
# fst x;;
- : int = 1
# snd x;;
- : int = 2
```

empty tuple, used instead of ‘void’

```
# ();;
- : unit = ()
# print_string;;
- : string -> unit = <fun>
```

function arguments

tupled form: like in an imperative language

```
# let diff (i,j) = if i < j then j-i else i-j;;
val diff : int * int -> int = <fun>
# diff (3, 4);;
- : int = 1
# (diff 3 4);;
This function is applied to too many arguments
```

curried form: stages the computation

```
# let diff i j = if i < j then j-i else i-j;;
val diff : int -> int -> int = <fun>
# (diff 3) 4;;
- : int = 1
# (diff 3 4);;
- : int = 1
```

lists

```
# [1;2];;
- : int list = [1; 2]
# 1::2::[];;
- : int list = [1; 2]
```

lists are homogeneous

```
# [[1]];;
- : int list list = [[1]]
# [1;[2]];;
This expression has type 'a list but is here used with type int
```

empty list is polymorphic

```
# [];;
- : 'a list = []
```

polymorphic functions

the simplest polymorphic function

```
# fun x -> x;;
- : 'a -> 'a = <fun>
```

a polymorphic function over lists

```
# let cons e l = e :: l;;
val cons : 'a -> 'a list -> 'a list = <fun>
# cons 1 2;;
This expression has type int but is here used with type 'a list
# cons 1 [];;
- : int list = [1]
```

datatypes

a simple datatype

```
# type color = Red | Green | Blue;;
type color = Red | Green | Blue
# Red;;
- : color = Red
# [Red ; Green];;
- : color list = [Red; Green]
```

constructors can take arguments

```
# type numtree = Empty | Tree of numtree * int * numtree;;
type numtree = Empty | Tree of numtree * int * numtree
# Empty;;
- : numtree = Empty
# Tree (Empty, 3, Empty);;
- : numtree = Tree (Empty, 3, Empty)
```

patterns

a function on number trees

```
# type numtree = Empty | Tree of numtree * int * numtree;;
# let rec treesum t =
  match t with Empty -> 0
  | Tree (t1, i, t2) -> i + treesum t1 + treesum t2;;
val treesum : numtree -> int = <fun>
# let tt = Tree (Tree (Empty, 1, Empty), 3, Tree (Empty, 2,Empty));;
…# treesum tt;;
- : int = 6
```

a function on lists

```
# let rec sum l = match l with [] -> 0 | e :: rest -> e + sum rest;;
val sum : int list -> int = <fun>
# sum [1;2;3;4];;
- : int = 10
```

puzzle: poly functional over lists

write the function map

- val map : ('a -> 'b) -> 'a list -> 'b list

solution

```
# let rec map f l =
    match l with [] -> [] | x :: xs -> (f x) :: (map f xs);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map dbl [1;2];;
- : int list = [2; 4]
```

puzzle: user-defined poly datatypes

a polymorphic tree

```
# type 'a tree = Empty | Tree of ('a tree) * 'a * ('a tree);;
type 'a tree = Empty | Tree of 'a tree * 'a * 'a tree
```

what is the type of treefold?

```
# let rec treefold f b t =
  match t with  Empty -> b
| Tree (left, v, right) -> f (treefold f b left, v, treefold f b right);;
```

```
val treefold : ('a * 'b * 'a -> 'a) -> 'a -> 'b tree -> 'a = <fun>
```

side-effects

mutable cells

```
# let seed = ref 0;;
val seed : int ref = {contents=0}
```

dereference

```
# !seed;;
- : int = 0
```

assignment

```
# seed := 1;;
- : unit = ()
# !seed;;
- : int = 1
```

puzzle

write a function *next*

- which produces 0, 1, 2, etc
- takes no arguments

closures and cells

```
# let next =
    (let seed = ref 0 in
      function () -> seed := !seed+1; !seed);;
val next : unit -> int = <fun>
# (next);;
- : unit -> int = <fun>
# next ();;
- : int = 1
# next ();;
- : int = 2
```

lazy lists or 'streams'

define a datatype for streams

```
# type 'a stream = Nil | Cons of 'a * (unit -> 'a stream);;
type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)

# let cons x s = Cons (x, fun () -> s);;
val cons : 'a -> 'a stream -> 'a stream = <fun>
# let hd s = match s with Cons (x,f) -> x;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: Nil
val hd : 'a stream -> 'a = <fun>

# let tl s = match s with Cons (x, f) -> f ();;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: Nil
val tl : 'a stream -> 'a stream = <fun>
```

using streams

```
# let rec from k = Cons (k, fun () -> from (k+1));;
val from : int -> int stream = <fun>
# (from 3);;
- : int stream = Cons (3, <fun>)
# hd (tl (from 3));;
- : int = 4
```

puzzle

given

```
# type 'a tree = Empty | Tree of 'a * 'a tree list;;
type 'a tree = Empty | Tree of 'a * 'a tree list
```

write a function that

- performs a depth-first traversal of a tree
- gives result as a stream

you can assume an infix function @

- for appending streams