Foundations of Computer Science
Lecture #4: More on Lists

Anil Madhavapeddy

2023-24

Warm-Up

Question 1a: What is the cost of evaluating `xs @ ys`?

O(List.length xs)

Question 1b: What is the cost of evaluating `x :: xs`?

O(1)

Question 2: What is the type of this function?

```
let rec flatten = function
  | []      -> []
  | l :: ls -> l @ flatten ls
```

```
Out: val flatten : 'a list list -> 'a list = <fun>
```

Question 3a: What does this return?

```
In [1]:    let a = [2];;
Out[1]:    val a : int list = [2]
In [2]:    let b = [3; 4; 5];;
Out[2]:    val b : int list = [3; 4; 5]
In [3]:    a::b;;
```

```
Error: This expression has type int list
       but an expression was expected of type int list list
       Type int is not compatible with int list
```

Question 3b: How to concatenate `a` and `b`?

```
In [4]:   a @ b;;
Out[4]:   - : int list = [2; 3; 4; 5]
```

Question 3c: Redefine `b` so that `a::b` works.

```
In [3]:   let b = [b];
Out[3]:   val b : int list list = [[3; 4; 5]]
In [4]:   a::b;;
Out[4]:   - : int list list = [[2]; [3, 4, 5]]
```

A Note on Notation

```
In :    let rec append1 = function
          | ([], ys)     -> ys
          | (x::xs, ys) -> x :: append1 (xs, ys)

Out:    val append : 'a list * 'a list -> 'a list = <fun>
```

```
In :    let rec append2 pair =
          match pair with
          | ([], ys)     -> ys
          | (x::xs, ys) -> x :: append2 (xs, ys)

Out:    val append2 : 'a list * 'a list -> 'a list = <fun>
```

A Note on Notation

```
In :    let rec append3 xs ys =
          match (xs, ys) with
          | ([], ys)    -> ys
          | (x::xs, ys) -> x :: append3 xs ys

Out:    val append3 : 'a list -> 'a list -> 'a list = <fun>
```

```
In :    let rec append4 xs ys =
          match xs with
          | []    -> ys
          | x::xs -> x :: append4 xs ys

Out:    val append : 'a list -> 'a list -> 'a list = <fun>
```

List Utilities: `take` and `drop`

$$xs = [\underbrace{x_0, \ldots, x_{i-1}}_{take(xs,\, i)}, \underbrace{x_i, \ldots, x_{n-1}}_{drop(xs,\, i)}]$$

List Utilities: `take` and `drop`

wildcard pattern

```
let rec take = function
| ([], _) -> []
| (x::xs, i) ->
    if i > 0 then
      x :: take (xs, i - 1)
    else
      []

let rec drop = function
| ([], _) -> []
| (x::xs, i) ->
    if i > 0 then
      drop (xs, i - 1)
    else
      x::xs
```

List Utilities: `take` and `drop`

```
Out:  val take : 'a list * int -> 'a list = <fun>
Out:  val drop : 'a list * int -> 'a list = <fun>
```

```
In :  let a = [1; 2; 3; 4; 5; 6];;
In :  take (a, 3);;
Out: - : int list = [1; 2; 3]

In:   drop (a, 3);;
Out:  - : int list = [4; 5; 6]
```

Linear Search

find x in list $[x_1, \ldots, x_n]$ by comparing with each element

obviously $O(n)$ TIME

simple & general

ordered searching needs only $O(\log n)$

indexed lookup needs only $O(1)$

more about search in Lecture 10...

Equality Tests

if (x=y) then true, else ...

```
let rec member x = function
| [] -> false
| y::l ->
    x = y || member x l
```

Equality testing is *OK* for integers but NOT for functions.

Equality Tests (cont.)

```
let rec inter xs ys =
  match xs, ys with
  | [], ys    -> []
  | x::xs, ys ->
      if member x ys then
        x :: inter xs ys
      else
        inter xs ys
```

Building a List of Pairs

$$\left.\begin{array}{l}[x_1, \ldots, x_n] \\ [y_1, \ldots, y_n]\end{array}\right\} \longmapsto [(x_1, y_1), \ldots, (x_n, y_n)]$$

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

Building a List of Pairs

```
let rec zip xs ys =
  match xs, ys with
  | (x::xs, y::ys) -> (x, y) :: zip xs ys
  | _ -> []
```

wildcard

The *wildcard* pattern (_) matches *anything.*

THE PATTERNS ARE TESTED IN ORDER.

For example, _ will match: `([], (y::ys))`

```
In :   zip [1;2;3;4] [‘a';'b';'c'];;
Out:   - : (int * char) list = [(1,'a'); (2,'b'); (3,'c')]
```

Building a List of Pairs

Two functions: **zip** and **unzip**

```
zip : 'a list -> 'b list -> ('a * 'b) list
unzip : ('a * 'b) list -> ('a list * 'b list)
```

Some Syntax

Expressions

$$\underline{\texttt{let}}\ D\ \underline{\texttt{in}}\ E$$

- Embeds declaration ***D*** within expression ***E***
- Useful within a function
- Can perform intermediate computations with function arguments

Building a Pair of Results

Version 1: With a local declaration.

```
           let rec unzip = function
           | [] -> ([], [])
           | (x, y)::pairs ->
declaration    let xs, ys = unzip pairs in
 expression    (x::xs, y::ys)
```

The `let` construct binds `xs` and `ys` to the results of the recursive call.

Example:

```
In :  unzip [(1,’a');(2,'b')];;
Out:  - : int list * char list = ([1; 2], ['a'; 'b'])
```

Building a Pair of Results

Version 2: Replacing local declaration by a function.

```
let conspair ((x, y), (xs, ys)) = (x::xs, y::ys)

val conspair :
  ('a * 'b) * ('a list * 'b list) ->
     'a list * 'b list = <fun>

let rec unzip = function
| [] -> ([], [])
| xy :: pairs -> conspair (xy, unzip pairs)
```

1 pair

list (of pairs)

pair of lists

Building a Pair of Results

Version 3: Iterative.

accumulators

```
let rec revUnzip = function
| ([], xs, ys) -> (xs, ys)
| ((x, y)::pairs, xs, ys) ->
    revUnzip (pairs, x::xs, y::ys)
```

Question: How to call `revUnzip`?

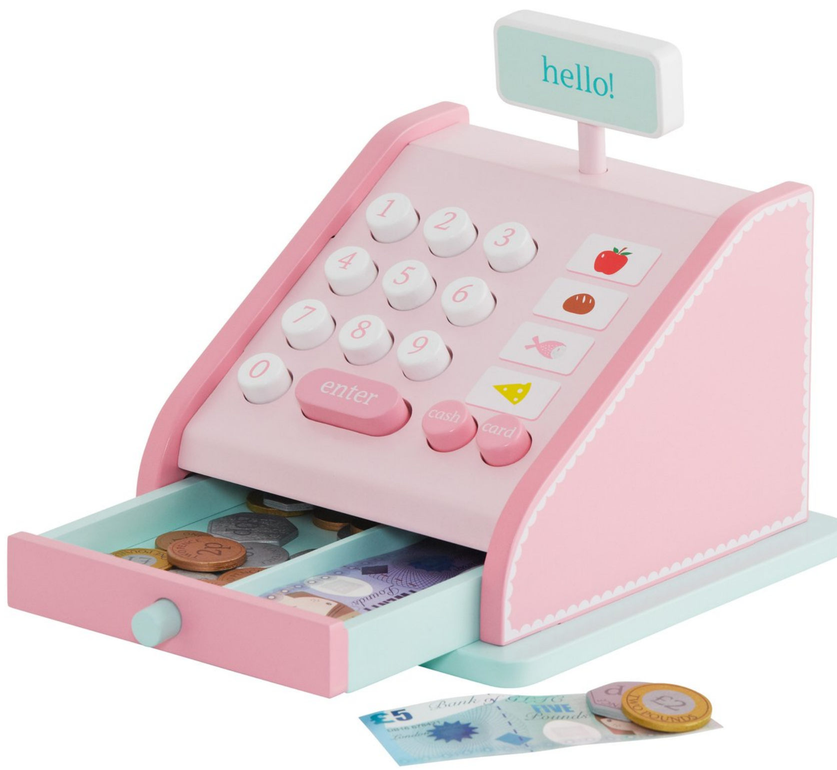
```
revUnzip (pairs, [], []);
```

Question: What's the result of the following?

```
let pairs = [("a", 1); ("b", 2)];;
revUnzip (pairs, [], []);;
```

```
Out: - : string list * int list = (["b"; "a"], [2; 1])
```

An Application: Making Change



- Till has unlimited supply of coins, for certain coin values
- List of coins `till` is given in descending order
- Larger coins preferred (tried first)

An Application: Making Change

list of possible coin values

```
let rec change till amt =
  if amt = 0 then
    []
  else
    match till with
    | [] -> raise (Failure "no more coins!")
    | c::till ->
        if amt < c then
          change till amt
        else
          c :: change (c::till) (amt - c)
```

- The recursion *terminates* when `amt` = 0.
- Tries the *largest coin first* to use large coins.
- The algorithm is *greedy*, and it CAN FAIL!

An Application: Making Change

```
let till = [50; 20; 10; 5; 2; 1];;
change till 43;;
```

~~50~~ 20 (amt=23) 20 (amt=3) ~~10~~ ~~5~~ 2 (amt=1) 1 (amt=0)

```
- : int list = [20; 20; 2; 1]
```

```
let till = [5; 2];;
change till 16;;
```

5 (amt=11) 5 (amt=6) 5 (amt=1) ~~2~~ ? amt≠0, till=[]

```
Exception: Failure "no more coins!"
```

An Application: Making Change

```
let rec change till amt =
  if amt = 0 then
    []
  else
    match till with
    | [] -> raise (Failure "no more coins!")
    | c::till ->
        if amt < c then
          change till amt
        else
          c :: change (c::till) (amt - c)
```

? $amt \neq 0$, $till = []$

ALL **Ways of Making Change**

Disclaimer: This is kind of hard.

```
let rec change till amt =
  if amt = 0 then
    [ [] ]
  else
    match till with
    | [] -> []
    | c::till ->
        if amt < c then
          change till amt
        else
          let rec allc = function
          | [] -> []
          | cs :: css -> (c::cs) :: allc css
          in
            allc (change (c::till) (amt - c)) @
                 change till amt
```

success (zero)

failure

generates all possible solutions

```
Out:  val change : int list -> int -> int list list = <fun>
```

ALL **Ways of Making Change**

...

declaration

```
let rec allc = function
| [] -> []
| cs :: css -> (c::cs) :: allc css
in
  allc (change (c::till) (amt - c)) @
       change till amt
```

int list

int list list

expression

@

use coin c

dont use coin c

c::[...], c::[...], ...

[...], [...], ...

cons c to solutions for amt-c

solutions for amt

ALL **Ways of Making Change**

```
In :  let till = [5; 3; 2];;
In :  change till 6;;

Out:  - : int list list = [[3; 3]; [2; 2; 2]]


In :  let till = [5; 2];;
In :  change till 16;;

Out:  - : int list list =
      [[2; 2; 2; 5; 5]; [2; 2; 2; 2; 2; 2; 2; 2]]
```

ALL **Ways of Making Change — Faster!**

accumulators

```
let rec change till amt chg chgs =
  if amt = 0 then
    chg::chgs
  else
    match till with
    | [] -> chgs
    | c::till ->
        if amt < 0 then
          chgs
        else
          change (c::till) (amt - c) (c::chg)
                 (change till amt chg chgs)
```

use coin

solutions that dont use coin

We've added **another accumulating parameter!**

Repeatedly improving simple code is called **stepwise refinement**.

ALL **Ways of Making Change — Faster!**

```
In :  change [5;3;2] 6 [] [];;

Out:  - : int list list = [[3; 3]; [2; 2; 2]]
```