Lecture 9 CSCI: Pipelining

Sebastiaan Joosten

September 25th

University of Minnesota

Driven to Discover®

Outlook

- Pipelining
- Currying

  - … and a bunch of program-transformation laws

Pipelining

- This short story is about a single function: |>

- let (|>) x f = f x
- So we can write: x |> f instead of f x

- Moreover, |> is left-associative:
- a |> b |> c = (a |> b) |> c

Ocaml-book’s ‘application’

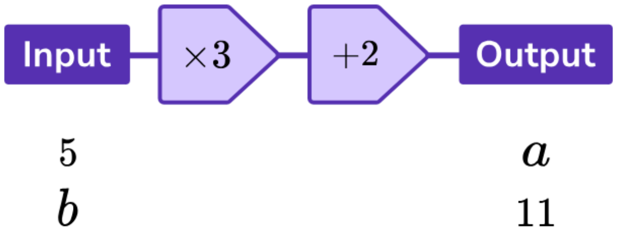
- We get to write fewer parentheses, compare these equivalent programs:

- `[1;2;3] |> List.map ((+) 2) |> List.iter print_int`
- `List.iter print_int (List.map ((+) 2) [1;2;3])`
- ```
  let x1 = [1;2;3] in
  let x2 = List.map ((+) 2) x1 in
  List.iter print_int x2
  ```
  ```

High school 'application'

- Easier to reason with.
- In this example: easier to compute the inverse

1. Find the missing Output and missing Input for the function machine.



- [ ] $a = 17,\ b = 35$
- [ ] $a = 21,\ b = 3$
- [ ] $a = 17,\ b = 3$
- [ ] $a = 13,\ b = 39$

Transcript:
Find the missing Output and missing input for the function machine
(a machine is drawn with the blocks Input, times 3, plus 2, Output connected in that order. Underneath the word Input is the number 5, underneath the word Output is the letter 'a', indicating a is the output if 5 is applied as input. Similarly, a 'b' is written further below Input, with a corresponding '11' below 'Output'.)
There are four multiple choice options for the answer

Combining the two …

- We’re often applying “transformed functions”, and it’s hard to reason about those.
- [1;2;3] |> List.map ((+) 2) |> List.iter print_int
- Here, List.map transforms (+) 2 into a function on lists

- [1;2;3] |> List.map ((+) 2) |> List.map ((*) 2)
- Here, we do two transformations, equivalent to:
- [1;2;3] |> List.map (fun x -> (x + 2) * 2)
- Why?

A bunch of other rules…

- [1;2;3] |> List.map ((+) 2) |> List.filter (fun x -> x < 5)
  = {why?}
  [1;2;3] |> List.filter (fun x -> x < 3) |> List.map ((+) 2)

A bunch of other rules…

- [1;2;3] |> List.map ((+) 2) |> List.filter (fun x -> x < 5)
  = {why?}
  [1;2;3] |> List.filter (fun x -> x + 2 < 5) |> List.map ((+) 2)
  = {simplification of x + 2 < 5}
  [1;2;3] |> List.filter (fun x -> x < 3) |> List.map ((+) 2)

A bunch of other rules…

- [1;2;3] |> List.map ((+) 2) |> List.filter (fun x -> x < 5)
  = {some map/filter property}
  [1;2;3] |> List.filter (fun x -> x + 2 < 5) |> List.map ((+) 2)
  = {simplification of x + 2 < 5}
  [1;2;3] |> List.filter (fun x -> x < 3) |> List.map ((+) 2)

A bunch of other rules…

- [1;2;3] |> List.map ((+) 2) |> List.filter (fun x -> x < 5)
  = {some map/filter property}
  [1;2;3] |> List.filter (fun x -> x + 2 < 5) |> List.map ((+) 2)
  = {simplification of x + 2 < 5}
  [1;2;3] |> List.filter (fun x -> x < 3) |> List.map ((+) 2)

up to integer overflows:

```
utop # max_int + 2 < 5;;
- : bool = true
utop # max_int < 3;;
- : bool = false
```

A bunch of other rules… (2)

- [1;2;3] |> List.filter (fun x -> x > 1) |> List.filter (fun x -> x < 3)
  = {why?}
  [1;2;3] |> List.filter (fun x -> x > 1 && x < 3)

Filter/map overview

- x |> List.filter p |> List.filter q
  = x |> List.filter (fun x -> p x && q x)
- x |> List.map f1 |> List.map f2
  = x |> List.map (fun x -> f2 (f1 x))
- x |> List.map f1 |> List.filter p
  = x |> List.filter (fun x -> p (f1 x)) |> List.map f1

- Conclusion: we can move reduce a map/filter pipeline into a call to ‘filter’ and then a call to ‘map’.

What about combining fold_right?

- Map and filter are instances of fold_right,
  so can we generalize our rules perhaps?

- x |> Map.fold_right f1 |> Map.fold_right f2 = …?

- Unfortunately, there's nothing that can be done without knowing more about f1

Currying

- Two useful helper functions:
  - let curry f x y = f (x,y)
  - let uncurry f (x,y) = f x y
- Note that the ‘f’ here has different types:

```
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Textbook’s reason behind Curry

Sometimes you will come across libraries that offer an uncurried version of a function, but you want a curried version of it to use in your own code; or vice versa

Why would you need an uncurried f ?

```
List.combine [1;2;3] ["a";"b";"c"]
 : (int * string) list
 = [(1, "a"); (2, "b"); (3, “c”)]

List.map2 (fun x y -> string_of_int x ^ ": " ^ y)
          [1;2;3]
          ["a";"b";"c"];;
 : string list = ["1: a"; "2: b"; "3: c”]

let map2 f l1 l2 = …
```

Why would you need an uncurried f ?

```
List.combine [1;2;3] ["a";"b";"c"]
 : (int * string) list
 = [(1, "a"); (2, "b"); (3, “c”)]

List.map2 (fun x y -> string_of_int x ^ ": " ^ y)
          [1;2;3]
          ["a";"b";"c"];;
 : string list = ["1: a"; "2: b"; "3: c”]

let map2 f l1 l2 = List.combine l1 l2 |> …
```

Why would you need an uncurried f ?

```
List.combine [1;2;3] ["a";"b";"c"]
 : (int * string) list
 = [(1, "a"); (2, "b"); (3, “c”)]

List.map2 (fun x y -> string_of_int x ^ ": " ^ y)
          [1;2;3]
          ["a";"b";"c"];;
 : string list = ["1: a"; "2: b"; "3: c”]

let map2 f l1 l2 = List.combine l1 l2 |>
                   List.map …
```

Why would you need an uncurried f ?

```
List.combine [1;2;3] ["a";"b";"c"]
 : (int * string) list
 = [(1, "a"); (2, "b"); (3, “c”)]

List.map2 (fun x y -> string_of_int x ^ ": " ^ y)
          [1;2;3]
          ["a";"b";"c"];;
 : string list = ["1: a"; "2: b"; "3: c”]

let map2 f l1 l2 = List.combine l1 l2 |>
                   List.map (uncurry f)
```

… a wild application of computations with pairs

- Consider taking the average of a list:
- let rec sum = function [] -> 0. | (h::tl) -> h +. sum tl
- let rec count = function [] -> 0. | (h::tl) -> 1 + count tl
- let average lst = sum lst /. float_of_int (count lst)

- What can we improve?

… a wild application of computations with pairs

- Consider taking the average of a list:
- let rec sum = function [] -> 0. | (h::tl) -> h +. sum tl
- let rec count = function [] -> 0. | (h::tl) -> 1 + count tl
- let average lst = sum lst /. float_of_int (count lst)

- Tail recursion: let’s use fold_left

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left …
- let count lst = fold_left …
- let average lst = sum lst /. float_of_int (count lst)

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count lst = fold_left …
- let average lst = sum lst /. float_of_int (count lst)

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count lst = fold_left (fun x _ -> 1 + x) 0 lst
- let average lst = sum lst /. float_of_int (count lst)

- What more can we improve?

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count lst = fold_left (fun x _ -> 1 + x) 0 lst
- let average lst = sum lst /. float_of_int (count lst)

- We iterate over the same list twice, in separate ‘loops’
- Let’s combine those!

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count lst = fold_left (fun x _ -> 1 + x) 0 lst
- let sumcount lst = fold_left ??
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

A rule for combining folds

- (List.fold_left f1 z1 lst, List.fold_left f2 z2 lst)
  =
  List.fold_left (fun (a1,a2) h -> (f1 a1 h, f2 a2 h)) (z1,z2) lst

- You need this rule in lab!

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count lst = fold_left (fun x _ -> 1 + x) 0 lst
- let sumcount lst = fold_left ??
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count_fn x _ = 1 + x
- let count lst = fold_left count_fn 0 lst
- let sumcount lst = fold_left ??
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count_fn x _ = 1 + x
- let count lst = fold_left count_fn 0 lst
- let combined_fn = fun (a1, a2) h -> (count_fn a1 h, a2+.h)
- let sumcount lst = fold_left …
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… a wild application of computations with pairs

- Consider taking the average of a list:
- let sum lst = fold_left (+.) 0. lst
- let count_fn x _ = 1 + x
- let count lst = fold_left count_fn 0 lst
- let combined_fn = fun (a1, a2) h -> (count_fn a1 h, a2+.h)
- let sumcount lst = fold_left combined_fn (0, 0.) lst
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… cleaning up …

- let count_fn x _ = 1 + x
- let combined_fn = fun (a1, a2) h -> (count_fn a1 h, a2+.h)
- let sumcount lst = fold_left combined_fn (0, 0.) lst
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… cleaning up …

- let count_fn x _ = 1 + x
- let combined_fn (a1, a2) h = (count_fn a1 h, a2+.h)
- let sumcount lst = fold_left combined_fn (0, 0.) lst
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

… cleaning up …

- let combined_fn (a1, a2) h = (a1 + 1, a2 +. h)
- let sumcount lst = fold_left combined_fn (0, 0.) lst
- let average lst = match sumcount lst with
  | (s, n) -> s /. n

- .. and here the partially uncurried combine_fn pops up

… cleaning up …

- let combined_fn (a1, a2) h = (a1 + 1, a2 +. h)
- let sumcount lst = fold_left combined_fn (0, 0.) lst
- let average lst = lst |> sumcount |> uncurry (/.)

Outlook

- Pipelining with option
- Modifiable input / output