An Introduction to Functional Programming

Intermediate Computer Science Pre-College Program
23-27 July 2018
– Module 2 –

Topics for Today

- Polymorphism
- Higher-order functions
- Lists – Basics
- Recursion and Pattern Matching on Numbers
- More Lists

Polymorphism

- What is the type of this function?

```
let id x = x
```

- Here are examples of its use:

```
# id 2;;
- : int = 2
# id true;;
- : bool = true
# id "hello";;
- : string = "hello"
```

- Its type should be `t -> t`, for any type `t`
- How do we express such a type? We use type variables

```
'a -> 'a
```

Polymorphism

```
let id x = x;;
val id : 'a -> 'a = <fun>
```

- This type is read as follows:

  `id` is a function that given a value of type `'a`, returns another value of the same type `'a`

- We say `id` is polymorphic
- Notice that
  - `id 2` has type `int` and
  - `id true` has type `bool`

Polymorphism

- It is a feature of type systems
- It allows an expression to have infinite types
- The type system then adjusts these types to more concrete ones depending on the use of these expressions
  - `id: 'a -> 'a (* general type *)`
  - `id: int -> int (* more concrete type *)`
- This style of polymorphism is called parametric polymorphism (the parameter is the type variable)

More Examples

```
# let f x = 7;;
```

- What does `f` do and what is its type?

```
# let f x = 7;;
val fst : ’a -> ’int = <fun>
```

More Examples

```
# let fst (x,y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# fst (2,3);;
- : int = 2
```

- `fst` takes a pair of type `'a * 'b` and returns a result of type `'a`

More Examples

```
# let f x y = x;;
```

- What does this function do?
- What is its type?

Exercise

- Write a function `swap` that takes in a pair and returns the same pair but where the components have been swapped
- For example, `swap (2,true)` should return `(true,2)`
- What is the type of this function?

Motivating Example

What is the type of the following function?

```
# let twice f x = f (f x);;
```

Consider the following example

```
# let twice f x  =  f (f x);;
val t : (’a -> ’a) -> ’a -> ’a = <fun>
# let sqr x =  x*x;;
val sqr: int -> int = <fun>
# twice sqr 2
- : int = 16
```

Higher-Order Functions

A function that takes another function as argument or that returns a function as result[1]

[1] The precise notion is more technical; this suffices for us.

Higher-Order Functions

Are these functions higher-order?

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let myAnd x y = x && y;;
val myAnd : bool -> bool -> bool = <fun>
```

- According to our definition, they are
  - `add`, given an integer, returns a function
- Note: `int -> int -> int` is the same as writing `int -> (int -> int)`
  - `->` associates to the right

Higher-Order Functions

A function that takes another function as argument or that returns a function as result[2]

Some more examples

```
# let apply f x = f x;;
val apply : (’a -> ’b) -> ’a -> ’b = <fun>
# apply (fun x -> (x,x)) 3;;
- : int * int = (3, 3)
# let apply’ =  fun f -> (fun x -> f x);;
val apply’ : (’a -> ’b) -> ’a -> ’b = <fun>
# apply’ (fun x -> (x,x)) 3;;
- : int * int = (3, 3)
```

[2] The precise notion is more technical; this suffices for us.

More Examples

```
# let compose f g x = f (g x);;
val compose : (’a -> ’b) -> (’c -> ’a) -> ’c -> ’b = <fun>
# let sqr x = x*x;;
val sqr : int -> int = <fun>
# compose sqr sqr 2;;
- : int = 16
```

Lists

A list is an ordered sequence of values of the same type

```
# [1;2;3];;
- : int list = [1; 2; 3]
# [1;1;3];;
- : int list = [1; 1; 3]
# ["hello"; "bye"];;
- : string list = ["hello"; "bye"]
# [1;"hello"];;
Error: This expression has type string but an expression was expected
of type int
# 1::[2;3];;
- : int list = [1; 2; 3]
# [];;
- : 'a list = []
# 1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

`::` is called cons, it adds an element to the beginning of a list

Lists – `cons` Operator

`::` is called cons

- it adds an element to the beginning of a list
- its type is `'a -> 'a list -> 'a list`

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# [];;
- : 'a list = []
# 1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

Lists – Append

```
# [1;2;3] @ [4;5];;
- : int list = [1; 2; 3; 4; 5]
# [1;2;3] @ [];;
- : int list = [1; 2; 3]
# [1;2] @ ["hello";"bye"];;
Error: This expression has type string but an expression was expected
of type int
```

Recall: use = for equality checking

Concatenating Lists

- Which of these are true and which are false?
- Under what assumptions?

```
[[]] @ xs   = xs
[[]] @ [xs] = [[],xs]
[[]] @ xs   = [xs]
[]::xs      = xs
[[]] @ [xs] = [xs]
[[]] @ xs   = []::xs
[xs] @ [xs] = [xs,xs]
[]   @ xs   = []::xs
[[]] @ xs   = [[],xs]
[xs] @ []   = [xs]
```

`List` Module

- Contains many useful operations on lists
- One example is `length`

```
# List.length [1;1;2];;
- : int = 3
# length [1;2;3];;
Error: Unbound value length
# open List;;
# length [1;2;3];;
- : int = 3
```

Note: Browsable sources of OCaml libraries

`http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/stdlib/`

Recursion

- Problem: Write a program that, given an integer n, adds the first n integers
- Example: if $n = 10$ then we want to add

$$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
val sum : int -> int = <fun>
# sum 0;;
- : int = 0
# sum 10;;
- : int = 55
```

Recursion

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
```

- `rec` says that we are defining a recursive function
    - A recursive function is a function that can call itself
- `match` is used for pattern matching on `n`
    - It is typically used in combination with `rec` but doesn't have to
- Lets follow the execution of couple of uses of `sum`

Recursion

On the board:

- `sum 0`
- `sum 1`
- `sum 2`
- `sum 3`

Recursion

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
# sum (-3);;
Stack overflow during evaluation (looping recursion?).
# let rec sum n =
  match n with
    0 -> 0
  | n when n>0 -> n + sum (n-1)
  | _ -> failwith "sum:: argument must be non-negative";;
val sum : int -> int = <fun>
# sum (-3);;
Exception: Failure "sum:: argument must be non-negative".
# sum 10;;
- : int = 55
```

Another Example of Recursion

- Problem: Write a program that, given an integer n, multiplies the first n integers
- Note: if $n = 0$ it should return 1
- Example: if $n = 10$ then we want to return

$$1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$$

```
# let rec fact n =
  match n with
    0 -> 1
  | n -> n * fact (n-1);;
val fact : int -> int = <fun>
# fact 0;;
- : int = 0
# fact 10;;
- : int = 3628800
```

Exercise

- Write a function `list_enum` that given a positive number `n` returns the list `[n;n-1;...;1;0]`
- For example, `list_enum 5` should return `[5;4;3;2;1;0]`
- What is the type of `list_enum`?

Exercise

- Write a function `repeat` that given an argument `x` and a positive number `n` returns the list `[x;x;...;x]` where `x` is repeated `n` times
- For example, `repeat "hello" 4` should return `["hello"; "hello"; "hello"; "hello"]`
- What is the type of `repeat`?

Exercise

- Write a function `stutter` that given two positive numbers `n` and `m` returns a new list of the form `[[n;n;...;n];[n-1;n-1;...;n-1];...;[0;0;...;0]` where each nested list has `m` items
- For example, `stutter 3 2` should return `[3;3];[2;2];[1;1];[0;0]]`
- What is the type of `stutter`?

The Length of a List

```
let rec length l =
  match l with
    [] -> 0
  | (x::xs) -> 1+ length xs
```

- Note the two cases in the definition:
  - the empty list `[]` – called the base case
  - the non-empty list `x::xs` – called the inductive case
- Run this function on a sample list
- What is the type of `length`?

Sum of a List of Numbers

```
let rec sum l =
  match l with
    [] -> 0
  | (x::xs) -> x + sum xs
```

Exercise

- Write a function that multiplies all the numbers in a list

Functions that Construct Lists

```
let rec incr l =
  match l with
    [] -> []
  | (x::xs) -> (x+1)::(incr xs)
```

Stutter

What does this function do?

```
let rec stutter l =
  match l with
    [] -> []
  | (x::xs) -> x::x::(stutter xs)
```

Exercise

- Define a function `is_zero_list` that given a list of numbers returns a list of booleans indicating whether each number is 0 or not.
- For example,

```
> is_zero_list [3;0;7;0;0];;
- : bool list = [false; true; false; true; true]
```

- What is the type of this function?

Functions that Filter Elements from a List

What does this function do?

```
let rec even l =
  match l with
    [] -> []
  | (x::xs) -> if (x mod 2=0) then x :: (g xs) else even xs
```

- Try it out on an example

Functions that Filter Elements from a List

What does this function do?

```
let rec even l =
  match l with
    [] -> []
  | (x::xs) -> if (x!=[]) then x :: (g xs) else even xs
```

- Try it out on an example

Functions that Filter Elements from a List

- Define a function that given a list of strings and a number `n`, filters (i.e. keeps) those strings whose length is smaller or equal to `n`
- What is the type of this function?

Functions on Lists that Deviate from Standard Patterns

- The standard patterns when defining a recursive function `f` over lists are:
  - Define `f` over the empty list `[]` (called base case)
  - Define `f` over the non-empty list `x::xs` (called inductive case)
- Some functions however don't fall in that scheme
- Here are some examples that we will develop on the board:
  - `head`
  - `tail`
  - `maximum`
  - `last`
  - `remove_adjacents`

Summary

- Polymorphism
- Higher-order functions
- Recursion on numbers and lists
- Functions on lists