Functional Programming

Li Shi

2020.9.17

What is functional programming?

- Imperative programming is based on statements.
- Functional programming··· is based on functions.
  - Functions are first-class citizens.

- Example:

C	OCaml

`int a[3] = {1, 2, 3};`
`for (int i = 0; i < 3; i++)`
`  a[i] *= 2;`
	`let double x = x * 2 in`
	`List.map double [ 1; 2; 3 ]`

Partial functions

- What is the meaning of the following OCaml code?

```
let plus a b =
  a + b
```

  - Type: `plus: int -> int -> int = <fun>`

- How about this one? Missing argument???

```
let plus2 =
  plus 2
```

  - Type: `plus2: int -> int = <fun>`

Currying

- `plus : int -> int -> int`
- `plus 2 : int -> int`
- `plus 2 3 : int`

- That's why we use the symbol "`->`" !

Feature 1. Pure functions

- A **pure function** is one without any **side-effects**.
  - Same input always generates the same output.
- Benefits:
  - Easy for compiler optimization
    - Remove unused functions
    - Store the result of a function used many times but with exactly the same input
  - Easy for scheduling
    - Order of functions can be reversed
    - Multi-thread safe
  - …

Tip 1: Avoid using array and pointer in FP!

Feature 2. Type inference

- Easy to perform static type checking and inference, such that potential type mistakes or confusion can be avoided, like adding an integer to a float.
- Example 1:

```
e1,e2 : int → e1+e2 : int
```

- Example 2:

```
e0 : bool, e1,e2 : T → if e0 then e1 else e2 : T
```

Tip 2: Use recursions rather than loop! (Why?)

Feature 3. Type matching on Datatypes

```
type expr =
  | Plus of expr * expr    (* means a + b *)
  | Minus of expr * expr   (* means a - b *)
  | Times of expr * expr   (* means a * b *)
  | Value of string        (* "x", "y", "n", etc. *)
```

```
let rec to_string e =
match e with
  | Plus  (left, right) -> "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
  | Minus (left, right) -> "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
  | Times (left, right) -> "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
  | Value v -> v
```

Feature 4. Pipe operators

- Pipe operators (`|>, <|`) will make your program more succinct and clearer.
- Example: Which one is the best?

```
let euclidean_distance v1 v2 =
  sqrt (List.fold_left ( +. ) 0. (List.map (fun x -> x *. x) (List.map2 ( -. ) v1 v2)))
```

```
let euclidean_distance v1 v2 =
  let v3 = List.map2 ( -. ) v1 v2 in
  let v4 = List.map (fun x -> x *. x) v3 in
  let v5 = List.fold_left ( +. ) 0. v4 in
  sqrt v5
```

```
let euclidean_distance v1 v2 =
  List.map2 ( -. ) v1 v2
  |> List.map (fun x -> x *. x)
  |> List.fold_left ( +. ) 0.
  |> sqrt
```

Exercise

- Read integers from keyboard until a zero. Store them in a list and print it.

Example Input	Example Output
1 2 3 4 0	[1; 2; 3; 4]

Reference

- OCaml Tutorials, ocaml.org/learn/tutorials/
- Real World OCaml by Y. Minsky, et al.