functional_programming.html

introduction.css

Functional Programming with OCaml {

< Tazmilur Saad >

}

functional_programming.html

introduction.css

1 2 4 4

#1

```
(1 * 2) 4 4
      → (2 * 4) 4
            → (8 * 4)
                  → 32
```

#2

```
1 2 4 4   1
1 2 4 4   2
1 2 4 4   8
1 2 4 4   32
```

recursion isn’t evil :)

* Church-Turing thesis ⇒ Recursion and iteration are equally expressive.
* Recursion is not necessarily expensive.
* Many problems can be understood much more easily from a recursive point of view

linked lists

* Homogenous type 'a
* Empty list → Nil
* [1] → (1 . Nil)
* [1 ; 2] → (1 . (2 . Nil))
* 1 :: 2 :: [] → [1;2]

multiply

```
let rec multiply l = match l with
    | [] → 1
    | h :: t → h * multiply t
```

* Pattern matching
* O(n) complexity but O(n) call stack :(

tail recursive multiply

```
let multiply l =
    let rec mul_aux l acc = match l with
        | [] → acc
        | h :: t → mul_aux t (h * acc)
in mul_aux l 1
```

O(n) time complexity and O(1) stack space :)

functional_programming.html

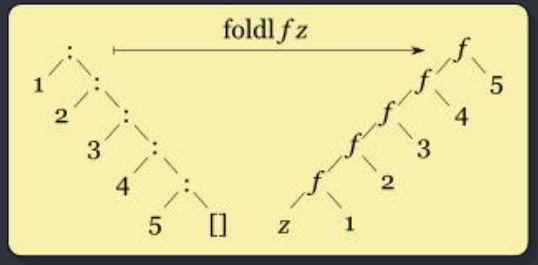
functions.css

foldl

* This is a common programming pattern called fold_left.
* List.foldl (*) 1 l
* Insertion sort is a fold!

```
let rec insert l e = match l with
| [] → [e]
| h :: t → if e > h then h :: insert t e else e :: l
```

* We will revisit fold multiple times :)



functional_programming.html

types.css

type inference

* Static typing
* Hindley-Milner type system.
* Received the Turing award for “ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism”
* Linear time type checking
* Never infers the wrong type
* Needs very little type annotation

robin milner (1934-2010)

but the best part is

* There is no null type. We will cover this later :)
* There is no any type. You don't have an easy-to-abuse escape hatch.
* Initial learning curve can be high but you will thank it later!

higher order functions

* Functions are **first-class** in OCaml.
* “[MapReduce] is inspired by the map and reduce primitives present in Lisp and many other functional languages” - Dean and Ghemawat, 2008.
* `val map : ('a → 'b) → 'a list → 'b list`

```
let double x = x * 2 in

    List.map double [1;2;3]
```

or even more concisely

```
List.map (fun x → x * 2) [1;2;3]
```

currying

* Every function in OCaml is unary.
* f: 'a → 'b → 'c ⟺ f: 'a → ('b → 'c)
* So if I have a function add x y = x + y, what happens if I evaluate add 1?

applications of currying

* You can use currying and fold to implement many other common functions!

```
let length lst = List.fold_left (fun acc _ → acc + 1) 0 lst

let rev lst = List.fold_left (fun acc x → x :: acc) [] lst

let map f lst = List.fold_right (fun x acc → f x :: acc) lst []
```

* There's also foldr -- you can check it out on your own :)

applications of currying

* If we want to test a merge_sorted function, we need to come up with some test cases.
* Here are some simple test cases!

```
let ms_empty = Alcotest.(check (list int)) "same lists" [] (merge_sorted [] [])

let ms_one_empty = Alcotest.(check (list int)) "same lists" [1] (merge_sorted [1] [])

let ms_one_list = Alcotest.(check (list int)) "same lists" [1;2] (merge_sorted [1] [2])

let ms_multi_list = Alcotest.(check (list int)) "same lists" [1;2;3;4] (merge_sorted [1;3] [2;4])
```

applications of currying

* We can refactor and reduce code repeats by using currying and partial application.

```
let binary_test_generator a b c f = Alcotest.(check (list int)) "same lists" a (f b c)
let merge_sorted_generator a b c = binary_test_generator a b c merge_sorted
let ms_empty () = merge_sorted_generator [] [] []
let ms_one_empty () = merge_sorted_generator [1] [1] []
let ms_single_list () = merge_sorted_generator [1;2] [1] [2]
let ms_multi_list () = merge_sorted_generator [1;2;3;4] [1;3] [2;4]
```

mutability

* Mathematical functions are **pure** -- they do not depend on any internal state and are not dependent on time.
* This allows composability and **referential transparency**.
* 1 + (2 * 3) ⟺ 1 + 6
* A pure language like Haskell does not have **side effects**. OCaml is not pure.
* OCaml uses **immutable** defaults.

immutability

* In some cases, immutability is required (i.e hashmaps)
* You don’t have to worry about race conditions :) Greatly simplifies concurrent and distributed programming.
* Is expensive.

mutability

* Mutable state comes with a lot of subtle gotchas.
* Mutability creates issues with aliasing.
* Mutability and side effects break referential transparency. It hides the change in its internal state and makes it harder to predict and reason about.
* Mutability increases the complexity of your code.

But the world is not pure :(

variants

* Sum types → something is x or y or z.
* Also known as discriminated unions or type safe unions.
* A popular feature from FP that has been recently included in many mainstream languages:
  - As std::variant in C++17 (2017)
  - As union types introduced in Python 3.5 (2014); upgraded in Python 3.10 (2019)
  - As union type in TypeScript

functional_programming.html

variant.css

I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language. [...] But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused the last forty years

- Tony Hoare, 1980 Turing Award Winner

options!

```
let rec find_double l a = match l with
    | [] → None
    | h :: t → if h = a then Some(2 * a) else find_double t a
```

How does this get rid of null exceptions?

* The return value of this function can never be used directly.
* Match requires that you exhaust all possibilities and is checked at compile time.

option is a variant

```
type 'a t = 'a option =
    | None
    | Some of 'a
```

* It is a sum type since it returns **either** None or Some.
* The Some branch contains data of type 'a
* Fundamentally different from C-style enums since they cannot carry any data

algebraic datatype

* ALGEBRA?
* Any datatype that is comprised of both sum types and product types is an algebraic data type, or ADT.

```
type 'a tree =
    | Leaf
    | Node of 'a * 'a tree * 'a tree
```

functions on trees

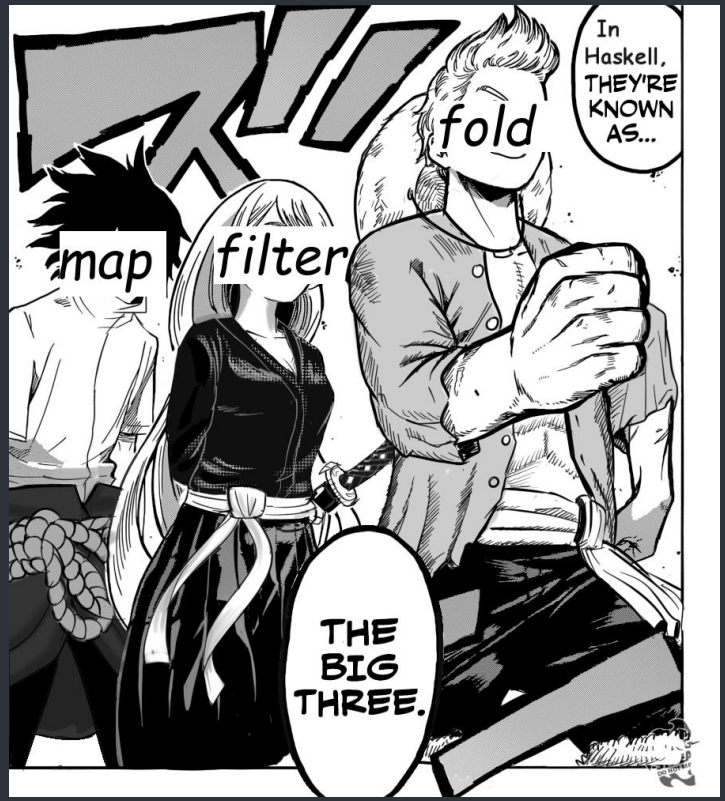
```
let rec fold_tree f acc = function
  | Leaf → acc
  | Node (v, l, r) → f v (fold_tree f acc l) (fold_tree f acc r)
let size t = fold_tree (fun _ l r → 1 + l + r) 0 t
let depth t = fold_tree (fun _ l r → 1 + max l r) 0 t
let preorder t = fold_tree (fun x l r → [x] @ l @ r) [] t
```

functional_programming.html

adt.css

functions on trees

* You can similarly define map, reduce, filter and other higher order functions on trees.



functional_programming.html

principles.css

principles of fp

* Minimize side effects and write predictable functions.
* Let an actual type system guide you and avoid errors.
* Reduce bloat and increase abstraction using higher order functions.
* Write elegant code :)

check out ...

OCaml is not the only functional language!

* Rust → Takes heavy inspiration from ML language family.
* Clojure → Lisp on the JVM.
* F# → Microsoft OCaml.
* Haskell → The only **pure** language.

You can also implement functional paradigms in your favorite languages!

thanks for coming!

Presentation created in consultation with:

1. Hickey, Jason, Anil Madhavapeddy, and Yaron Minsky. Real World Ocaml. 2nd ed. O'Reilly, 2021.
2. Clarkson, Michael R. OCaml Programming: Correct + Efficient + Beautiful, 2021.
3. Why OCaml. YouTube, 2016. https://www.youtube.com/watch?v=v1CmGbOGb2I
4. Foldl image courtesy of wikipedia
5. Programming Languages Laboratory @ JHU

functional_programming.html

currying.css

```
let merge_sorted a b =
  let rec merge_sorted_aux a b acc = match a with
  | [] → List.rev acc @ b
  | h :: t → match b with
    | [] → List.rev acc @ a
    | x :: y → if (h < x) then merge_sorted_aux t b (h :: acc)
                          else merge_sorted_aux a y (x :: acc)
  in merge_sorted_aux a b []
```