Lecture 3

Tuesday, August 31, 2021 1:57 PM

- Introduction to Types

Q1: How can we "construct" data?

Q2: How can we "destruct" data?

Nothing to do with "destructors" from OOP / C++.

~foo()

- Pairs, tuples, variants, records, lists
- Pattern matching
- ADTs & recursion

---

① Pairs & tuples

int * string

"int-cross-string"

fst : 'a * 'b → 'a

snd : 'a * 'b → 'b

"accessors"

(3, "Hello") : constructor

Q: How to access the elements of a pair?

"Destructor" / pattern matching

let myfst p = let (P1, P2) = P in P1.

Bind P1 to the first component

Bind P2 to the second component of P

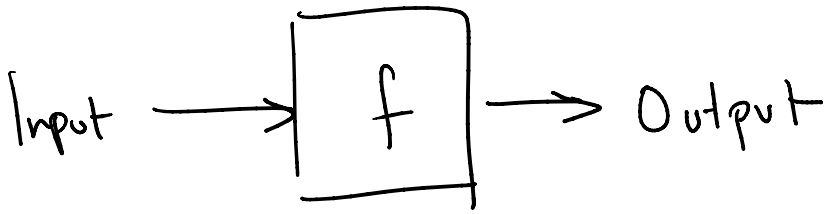
the first component of p

second component of p

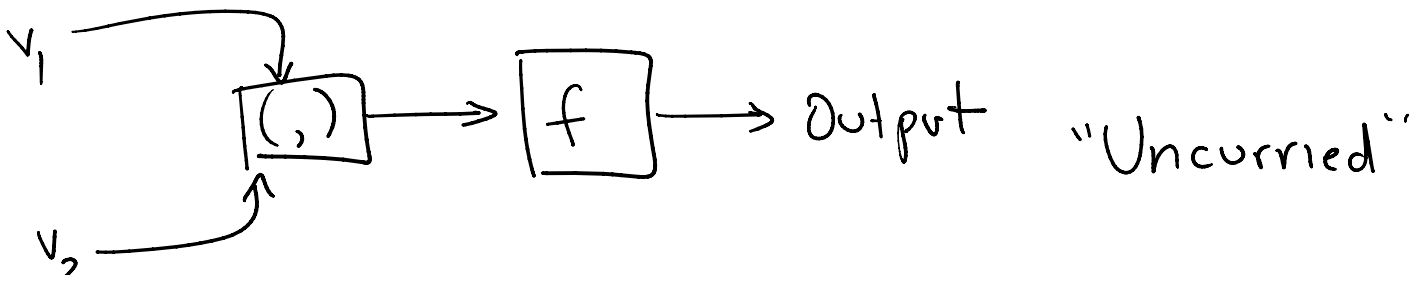
The type inference engine figures out that p has to be of type 'a * 'b.

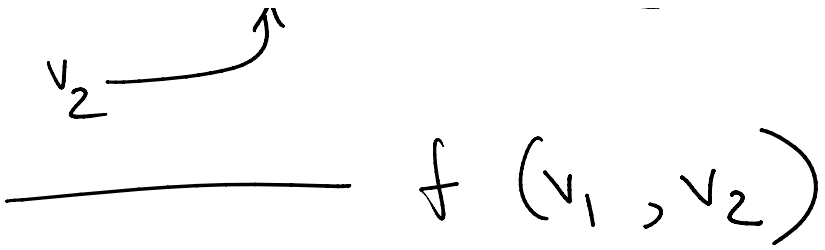
"It has to have the shape of a pair".

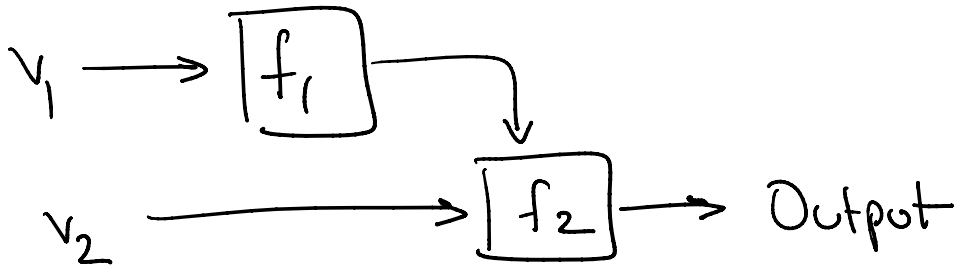
Technically, every Ocaml fn. takes a single value as input & produces a single value as output



How to devise fns with two inputs?







"Curried"

— let $f_2 = f_1\ v_1$ in

$f_2\ v_2$

Lists : [3; 4; 5] : "Syntactic sugar"

"Constructors"

- [] → Empty list
- a :: l → Take an element a & stick it in front of a list l.

$$[3; 4; 5] = 3 :: (4 :: (5 :: []))$$

Checking whether a list is empty.

```
let is_empty l = (List.length l) = 0
```

① Who defines List.length?

② Performance is O(n)

```
let is_empty l =
  match l with
  | [] -> true
  | hd :: tl -> false
```

Alternatives.

Replace with _ if hd not used.

Replace with _ if tl not used

Records

```
type person = { name : string;
                dob : int }
```

Shape of data

```
let p = { name = "Mukund" ; dob = 4897 };
```

Constructor

p.name , p.dob : Accessors

```
- let { name = pname ; dob = pdob } = p
  in pname
```

Destructor

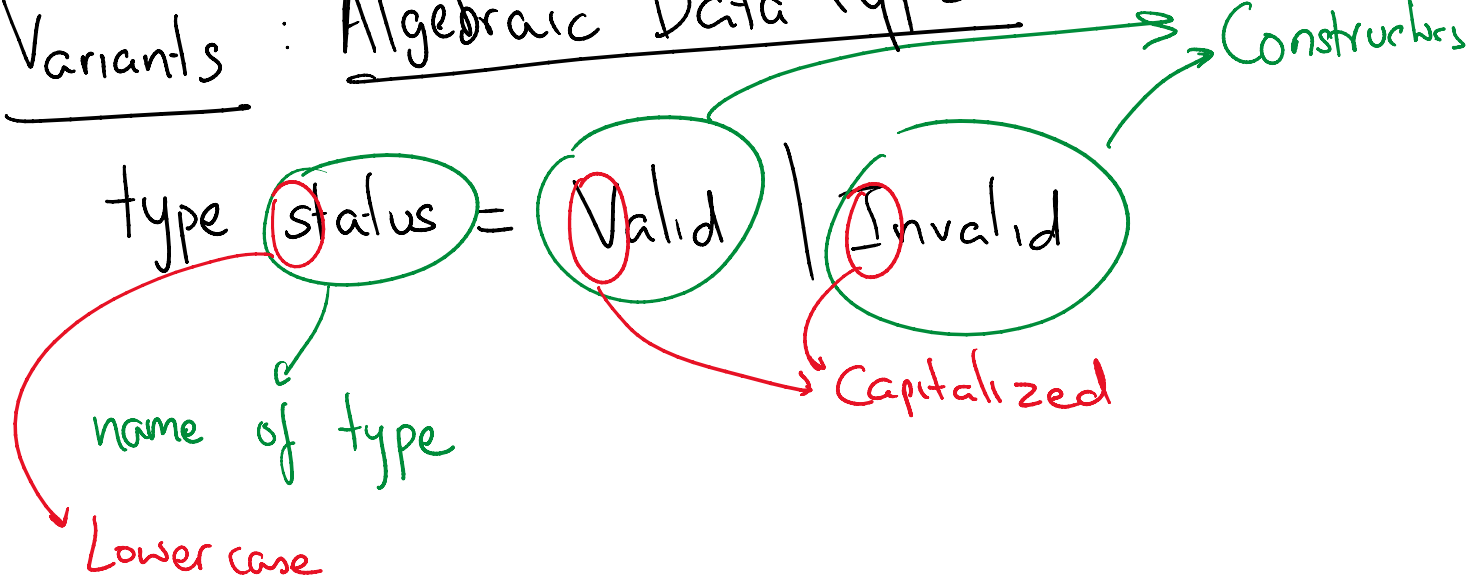
```
- let { name = pname ; _ } = p in pname

- let { name = pname; } = p in pname
```

warnings "+9" : Enables exhaustiveness check

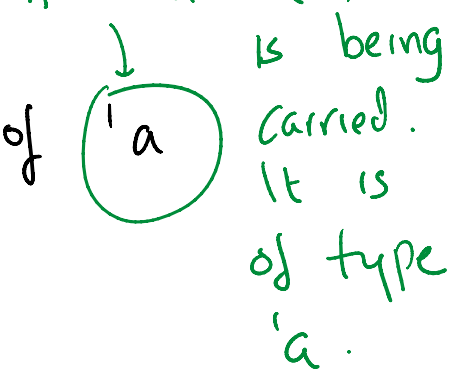
Algebraic Data Types

Variants : Algebraic Data Types

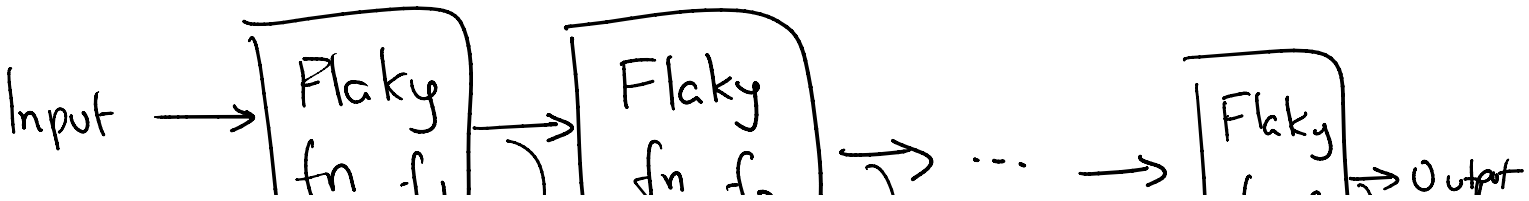


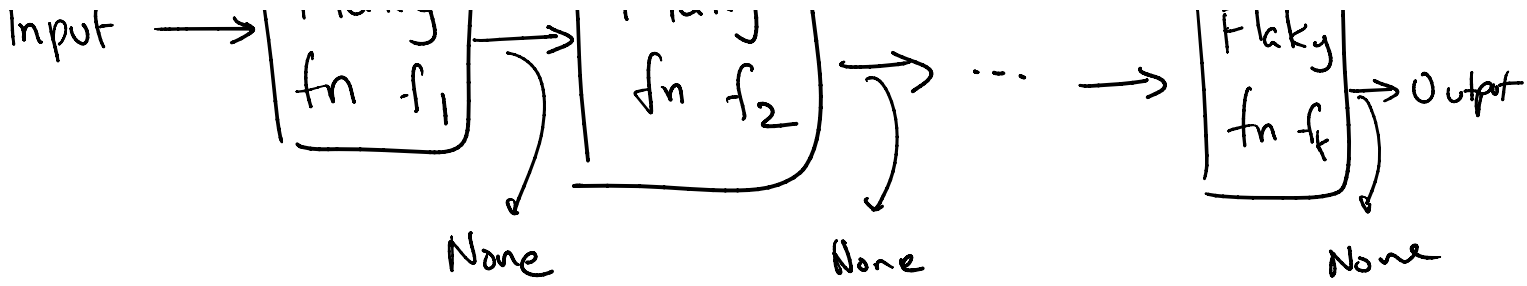

```
match s with
| Valid -> true
| Invalid -> false
```

Option types




```
type 'a option = Some of 'a
               | None
```





"Maybe" monad

Complex numbers : $a + ib$

$r(\cos\theta + i \sin\theta)$

Cartesian

Polar

```
type complex =
  | Cartesian of int * int
  | Polar of int * int
```

There are 8×10^{18} integers

64×10^{36} ways of building Cartesian

64×10^{36} ways

64×10 ways of building Polar complex #s

$\times$ ways of building Cartesian complex #s.

Totally, 128×10^{38} complex numbers in all.