OCaml: Recursive Types

Programming Languages

William Killian

Millersville University

Preface: Variants

- Variants allow us to make a **choice** between states
- These states:
  - Have names (called ***discriminators***)
  - Can be inspected with a match expression
  - Can optionally hold any value type specified
  - Can be created by specifying the discriminator first, followed by an expression that evaluates to its type.

```
type rank =
  Ace | King | Queen | Jack | Num of int
```

Extending Variants

- What if the **type** specified by a discriminator was the same type as the variant?

```
type magic =
  Nothing | Something of magic
```

Valid values could be:

- `Nothing`
- `Something (Nothing)`
- `Something (Something (Something (Nothing)))`

Recursive Types

- A type is **recursive** if in its implementation it specifies its own type as a storage unit.
  - In OCaml, this means that the type is used as a value type holder in one or more discriminators

```
type t = N | S of t
```

- What are some types you've worked with in other classes that might be recursive?

Lists

OCaml lists are recursive!

```
type 'a lst =
  Nil | Cons of 'a * 'a lst

(* Note: Not “real” OCaml *)
let (::) elem rest = Cons (elem, rest)
let [] = Nil
```

Natural Numbers

```
type nat = Zero | Succ of nat
```

We can model all natural (>= 0) numbers!

How can we represent 0?

1?

2?

10?

Natural Numbers

```
type nat =
    Zero
  | Succ of nat
```

Recursive Types => Recursive Functions!

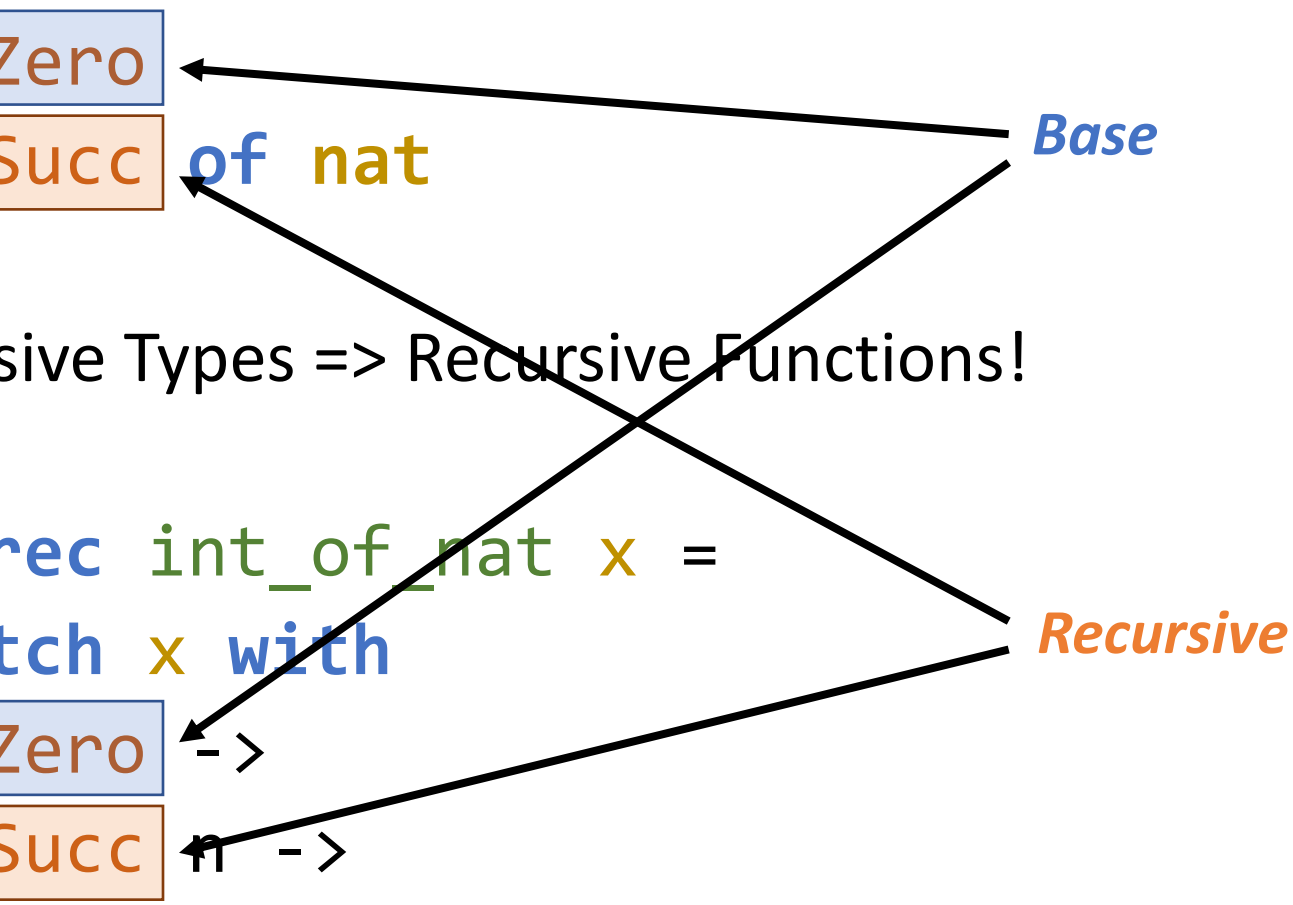
```
let rec int_of_nat x =
  match x with
    Zero ->
  | Succ n ->
```

Natural Numbers

```
type nat =
    Zero
  | Succ of nat
```

Recursive Types => Recursive Functions!

```
let rec int_of_nat x =
  match x with
    Zero -> 0
  | Succ n -> 1 + int_of_nat n
```



Natural Numbers

```
type nat =
    Zero
  | Succ of nat
```

Recursive Types => Recursive Functions!

```
let rec int_of_nat x =
  match x with
    Zero -> 0
  | Succ n -> 1 + (int_of_nat n)
```

Natural Numbers

```
type nat =
    Zero
  | Succ of nat
```

Recursive Types => Recursive Functions!

```
let rec nat_of_int n =
  if n == 0 then

  else
```

Natural Numbers

```
type nat =
    Zero
  | Succ of nat
```

Recursive Types => Recursive Functions!

```
let rec nat_of_int n =
  if n == 0 then
    Zero
  else
    Succ (nat_of_int (n - 1))
```

Natural Numbers: Addition

```
type nat =
    Zero
  | Succ of nat

let rec plus a b = (* Ideas? *)
```

Natural Numbers: Addition

```
type nat =
    Zero
  | Succ of nat

let rec plus a b =
  match b with
    Zero -> a
  | Succ b' -> Succ (plus a b')
```

Natural Numbers: Multiplication

```
type nat =
    Zero
  | Succ of nat

let rec times a b = (* Ideas? *)
```

Natural Numbers: Multiplication

```
type nat =
    Zero
  | Succ of nat

let rec times a b =
  match b with
    Zero -> Zero
  | Succ b' -> plus a (times a b')
```

List Operations

```
(* return the length of a list *)
let rec length l =
```

List Operations

```
(* return the length of a list *)
let rec length l =
  match l with
    [] -> 0
  | _::l' -> 1 + (length l')
```

List Operations

```
(* return the max element of a list *)
let rec max l =
```

List Operations

```
(* return the max element of a list *)
let rec max l =
  let rec helper v lst =
    match lst with
      [] -> v
    | e::l' ->
      helper (if e > v then e else v) l'
  in
  let e::l' = l in
  helper e l'
```

List Operations

```
(* adds all elements of l2 to the end
   of l1, keeping elements in order  *)
let rec append l1 l2 =
```

List Operations

```
(* adds all elements of l2 to the end
   of l1, keeping elements in order  *)
let rec append l1 l2 =
  let rec helper a b =
    match b with
      [] -> a
    | e::b' -> helper (e::a) b'
  in
  rev (helper (rev l1) l2)
```

Trees

- How can we represent a binary tree?

```
type node =
```

Trees

- How can we represent a binary tree?

```
type node =
     Node of int * node * node
   | Null
```

Trees: Sum of All Nodes

```
type node =
    Node of int * node * node
  | Null

let rec sum n =
```

Trees: Sum of All Nodes

```
type node =
    Node of int * node * node
  | Null

let rec sum n =
  match n with
    Node (v,l,r) -> v + (sum l) + (sum r)
  | Null -> 0
```

Expressions

- I want to write a calculator!

- 4.0 + 2.9 ==> 6.9
- 512 - 92 ==> 420
- (4.0 + 2.9) * (512 - 92) - 878 ==> 2020

What type should I use for **expr**?

Expressions

```
type expr =
    Num of float
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Div of expr * expr
```

Evaluating Expressions?

```
let rec eval e =
  match e with
    Num x ->
  | Add (a,b) ->
  | Sub (a,b) ->
  | Mul (a,b) ->
  | Div (a,b) ->
```

Evaluating Expressions?

```
let rec eval e =
  match e with
    Num x -> x
  | Add (a,b) -> (eval a) +. (eval b)
  | Sub (a,b) -> (eval a) -. (eval b)
  | Mul (a,b) -> (eval a) *. (eval b)
  | Div (a,b) -> (eval a) /. (eval b)
```

String Representation?

```
let rec string_of_expr e =
  match e with
    Num x ->
  | Add (a,b) ->
  | Sub (a,b) ->
  | Mul (a,b) ->
  | Div (a,b) ->
```

String Representation?

```
let rec soe e =
  match e with
    Num x -> string_of_float x
  | Add (a,b) -> "(" ^ (soe a) ^ "+" ^ (soe b) ^ ")"
  | Sub (a,b) -> "(" ^ (soe a) ^ "-" ^ (soe b) ^ ")"
  | Mul (a,b) -> "(" ^ (soe a) ^ "*" ^ (soe b) ^ ")"
  | Div (a,b) -> "(" ^ (soe a) ^ "/" ^ (soe b) ^ ")"
```