OCaml: Variants

Programming Languages

William Killian

Millersville University

Core OCaml Datatypes

- Primitives

  `int` `float` `string` `bool`

- Aggregates:

  `'a list` *`tuple`* (product type)

- What's missing?

  sum types

Variants

- Also known as ***discriminated unions***
- Must provide a named label for each option
- Only **one** can be active at any time
- Examples:

```
type suit =
  Spades | Hearts | Clubs | Diamonds

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

Variants

- They can be used to define a **new type** and possible range of values for that type:

```
type suit =
  Spades | Hearts | Clubs | Diamonds
```

- They can optionally hold information.
  - We will have to add **`of <type>`** to each choice which can hold additional information

```
type int_option =
  Some of int | None
```

Variant Behavior

```
type suit =
  Spades | Hearts | Clubs | Diamonds
```

- When we have an expression of type `suit`, it can only hold one of four possible choices:
  - `Spades`, `Hearts`, `Clubs`, and `Diamonds`
- The tag, or **discriminator**, tells OCaml what choice we want to currently select
- The discriminator must always start with a capital letter. OCaml will yell at us otherwise.

Variant Behavior

- You can view each choice as a box.
- By default, each box (or choice) will be empty
- When a discriminator includes a storage specifier (denoted with **`of <type>`**), then the box will hold a value of the specified type

```
type int_option = Some of int | None
let x = None
let y = Some 5
(* x and y both have type int_option *)
```

Using Variants

```
type suit =
  Spades | Hearts | Diamonds | Clubs
type rank =
  Ace | King | Queen | Jack | Num of int
(* suit and rank are variants *)

type card = rank * suit

let top_of_deck : card = (Ace, Spades)
let bottom_of_deck : card = (Num 2, Clubs)
```

Using Variants

- But what if I wanted to inspect a variant?

```
let top = List.hd (shuffle deck)
print_card top
```

- How can we write `print_card`?
  - Need to inspect the rank
  - Need to inspect the suit
- Ideas?

Pattern Matching

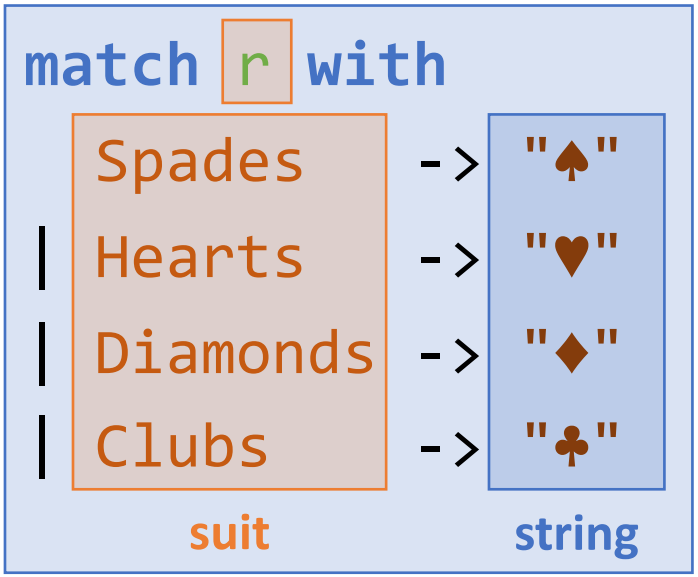
```
type suit =
  Spades
| Hearts
| Diamonds
| Clubs
```

```
match r with
  Spades -> "♠"
| Hearts -> "♥"
| Diamonds -> "♦"
| Clubs -> "♣"
```

- For **each discriminator**, add a match case.
- All expressions for the match must result in the **same type**

Pattern Matching

```
type suit =
   Spades
 | Hearts
 | Diamonds
 | Clubs
```



- For **each discriminator**, add a match case.
- All expressions for the match must result in the **same type**

Pattern Matching

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

```
match r with
  Ace    -> "Ace"
| King   -> "King"
| Queen  -> "Queen"
| Jack   -> "Jack"
| Num x -> string_of_int x
```

The type of x is defined by the type specified in the discriminator

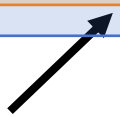
- For **each discriminator**, add a match case.
- All expressions for the match must result in the **same type**

Pattern Matching

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

rank string

```
match r with
  Ace    -> "Ace"
| King   -> "King"
| Queen  -> "Queen"
| Jack   -> "Jack"
| Num x  -> string_of_int x
```



The type of x is defined by the type specified in the discriminator

- For **each discriminator**, add a match case.
- All expressions for the match must result in the **same type**

Problem

- I want to have a list of int, float, bool, and string
- But OCaml is yucky and I can't do that…

… unless I use variants!

Solution Procedure:

1. Define a type that can hold all the types I need
2. Write appropriate helper methods
   - string_of

Defining a Type

- We need to hold: int, float, bool, and string

```
type box =
   Int    of int
 | Float  of float
 | Bool   of bool
 | String of string
```

Using the Type

```
let my_list = [
  Int 4;
  Float 1.2;
  Bool true;
  String "no-u";
  Int 6;
]
```

`my_list` has type **`box list`**

where each element is of type **`box`**

Using the Type

`my_list` has type **`box list`**

where each element is of type **`box`**

I should write a `string_of` function which accepts a **`box`** value and returns the string representation

```
let string_of v = ...
(* with types specified *)
let string_of (v:box) : string = ...
```

Using the Type

```
let string_of v =
  match v with
   | Int i    -> string_of_int i
   | Float f  -> string_of_float f
   | Bool b   -> string_of_bool b
   | String s -> s
```

Using the Type

```
let string_of = fun v ->
  match v with
   | Int i    -> string_of_int i
   | Float f  -> string_of_float f
   | Bool b   -> string_of_bool b
   | String s -> s

(* rewritten using fun *)
```

Matching Functions

The code pattern of:

```
fun v -> match v with
```

is so common, there is a special abbreviation syntax

```
function
```

where the argument name is completely omitted

- Match expression rules still apply
- All cases must be elaborated
- All cases must return the same type

Matching Functions

```
let string_of = function
  | Int i    -> string_of_int i
  | Float f  -> string_of_float f
  | Bool b   -> string_of_bool b
  | String s -> s

let as_string =
  List.map string_of my_list
(* ["4"; "1.2"; "true"; "no-u"; "6"] *)
```

Recap

- We can define our own **discriminated union** type when we want to choose between options:

```
type t = C1 | C2| C3
```

- Each choice can optionally hold a value.

```
type t = C1
       | C2 of int
       | C3 of string * float
```

- Must use pattern matching / match … with when extracting information
- Function is shorthand for

```
fun x -> match x with
```