Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 3 - Sequence 0: Tagged values

Overview of Week 3

1. Tagged values
2. Recursive types
3. Tree-like values
4. Case study: a story teller
5. Polymorphic algebraic datatypes
6. Advanced topics

Avoiding meaningless values

- In the database example, we had:

```
engine : database -> query -> status * database * contact
```

- The returned `database` and `contact` are meaningful only if the returned status is `true`. Otherwise, they must not be used.
- What if a type could capture this constraint?

Sum type: disjoint union of types

- We should change the type of `engine` into

```
engine : database -> query -> query_result
```

- Such that a value of type `query_result` can be **either**:
  - an error, or
  - a new database (in case an insertion or deletion query was successfully applied), or
  - a contact and its index (in case a search query was successful).
- In *OCaml*, this is written as a **sum type**:

```
type query_result =
| Error
| NewDatabase of database
| FoundContact of contact * int
```

Sum type: disjoint union of types

```
type some_type_identifier =
| SomeTag of some_type * ... * some_type
| ...
| SomeTag of some_type * ... * some_type
```

- `SomeTag` is a tag identifier, start with an uppercase letter.
- Tag identifiers must be unique and distinct.
- **A tag characterizes one specific type** in this disjoint union of types.
- How to **construct** and **observe** values of this type?

Constructing tagged values

- Tags are also called **constructors**.
- A tag is used as **a marker to classify values** with respect to the different cases of the union.

```
SomeTag (some_expression, ..., some_expression)
```

- Parentheses can be omitted if there is only one argument and if that argument is a simple expression (like a variable or a literal for instance).

A sum type for queries I

```
type query =
  | Insert of contact
  | Delete of contact
  | Search of string;;
```

```
# type query =
    Insert of contact
  | Delete of contact
  | Search of string
```

A sum type for queries II

```
let luke = { name = "luke"; phone_number = (1, 2, 3, 4) }
let query1 = Insert luke;;
# val luke : contact =
  {name = "luke"; phone_number = (1, 2, 3, 4)}
val query1 : query =
  Insert {name = "luke"; phone_number = (1, 2, 3, 4)}
let query2 = Search "luke";;
# val query2 : query = Search "luke"
let query3 = Delete luke;;
# val query3 : query =
  Delete {name = "luke"; phone_number = (1, 2, 3, 4)}
```

A special case: Tags with no argument are enumerations I

```
type color = Black | Gray | White;;
# type color = Black | Gray | White
let batman_s_color = Black;;
# val batman_s_color : color = Black
```

Observing tagged values by case analysis

- Let **`type`** `t = A | B` and `x` be an identifier of type `t`.
- `x` must have been constructed using `A` or `B`.
- Then, we know that `x` can be either an `A` **or** a `B`.
- If we want to write some computation that depends on `x`, we must provide some code for the case `x = A` and some code for the case `x = B`.

Observing tagged values by case analysis

- When we observe a value of a sum type, **several cases are possible**.
- The programmer must provide an **expression for each possible case**.
- A **case** is described by a pattern of the form:
  `SomeTag (some_pattern, ..., some_pattern)`
- A **branch** is composed of a pattern and an expression separated by an arrow.
  `some_pattern -> some_expression`

Case analysis by pattern matching

- A **pattern matching** is a sequence of branches:

```
match some_expression with
| some_pattern -> some_expression
| some_pattern -> some_expression
| ...
| some_pattern -> some_expression
```

- There must be **at least one branch** in a pattern matching.
- To evaluate a pattern matching:
  1. we compute the value of `some_expression`;
  2. we try to **match** it with the pattern of the first branch ;
  3. if it does not, we try the next one until we find a match.

A pattern matching

```
let engine db query =
  match query with
    | Insert contact -> insert db contact
    | Delete contact -> delete db contact
    | Search name -> search db name;;
```

Functions defined by cases

- Many functions start with a case analysis over one argument:

```
let f x = match x with
  | some_pattern -> some_expression
  | ...
  | some_pattern -> some_expression
```

- There is syntactic shortcut to define them:

```
let f = function
  | some_pattern -> some_expression
  | ...
  | some_pattern -> some_expression
```

A function defined by cases

```
let engine db query = function
    | Insert contact -> insert db contact
    | Delete contact -> delete db contact
    | Search name -> search db name;;
```

In the machine

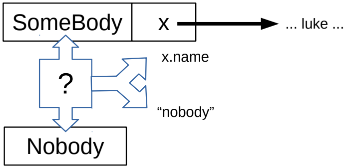
Program

let p =Somebody luke

let q = Nobody

function
| Somebody x → x.name
| Nobody → “nobody”

Machine



- Each tag is represented by a (small) machine integer.
- A value of a sum type is either:
  - a tag if it is a constructor with no argument;
  - or a heap-allocated block starting with a tag.
- Pattern matching performs a dynamic test on this tag.

Pitfalls

- A pattern can be ill-typed.
- A case analysis can be non exhaustive.
- All these programming errors are caught by the type-checker!

Ill-typed patterns I

```
type data = None | Single of int | Pair of int * int;;
# type data = None | Single of int | Pair of int * int
let bad_arity (x : data) =
  match x with
    | None x -> x
    | Single x -> x
    | Pair (x, _) -> x;;
# Characters 48-54:
      | None x -> x
        ^^^^^^
Error: The constructor None expects 0 argument(s),
       but is applied here to 1 argument(s)
```

Ill-typed patterns II

```
let bad_argument_type (x : data) =
  match x with
    | Single true -> false
    | _ -> true;;
# Characters 63-67:
      | Single true -> false
               ^^^^
Error: This pattern matches values of type bool
       but a pattern was expected which matches values of type int
```

Non exhaustive case analysis I

```
type color = Black | Gray | White;;
# type color = Black | Gray | White
(* Black < Gray < White and forall x, not (x < x). *)
let lighter c1 c2 =
  match (c1, c2) with
    | (Black, Black) -> false
    | (White, White) -> false
    | (Gray, Gray) -> false
    | (Black, _) -> true
    | (_, White) -> true
    | (White, Gray) -> false
    | (Gray, Black) -> false;;
```

Non exhaustive case analysis II

```
# Characters 76-291:
  ..match (c1, c2) with
      | (Black, Black) -> false
      | (White, White) -> false
      | (Gray, Gray) -> false
      | (Black, _) -> true
      | (_, White) -> true
      | (White, Gray) -> false
      | (Gray, Black) -> false..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(White, Black)
val lighter : color -> color -> bool = <fun>
```