Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 2 - Sequence 2: Constructing and Observing Records

Naming components

- The role of each component of a tuple is determined by its position.
- It is easy to use a wrong index.
- What if we could **name** components?

2D points as records I

```
type point2D = { x : int; y : int };;
# type point2D = { x : int; y : int; }
let origin = { x = 0; y = 0 };;
# val origin : point2D = {x = 0; y = 0}
let from_tuple (x, y) = { x; y };;
# val from_tuple : int * int -> point2D = <fun>
let a : point2D = from_tuple (4,  2);;
# val a : point2D = {x = 4; y = 2}
let b : point2D = from_tuple (10, 5);;
# val b : point2D = {x = 10; y = 5}
```

2D points as records II

```
type box = {
  left_upper_corner  : point2D;
  right_lower_corner : point2D;
};;
# type box = {
  left_upper_corner : point2D;
  right_lower_corner : point2D;
}
let the_box = { left_upper_corner = a; right_lower_corner = b };;
# val the_box : box =
  {left_upper_corner = {x = 4; y = 2};
   right_lower_corner = {x = 10; y = 5}}
let get_min_x { left_upper_corner  = { x } } = x;;
# val get_min_x : box -> int = <fun>
```

Syntax to declare a record type

- Contrary to tuples, record types must be declared.
- To declare a record type:

```
type some_type_identifier =
  { field_name : some_type; ...; field_name : some_type }
```

- All field names must be distinct.
- (And preferably unused in other record types.)

Syntax to construct a record

- To construct a record:

```
{ field_name = some_expression; ...; field_name = some_expression }
```

Syntax to observe a record

- To observe a specific field:

```
some_expression.field_name
```

- To observe several fields of a record, one can use **record patterns**:

```
{ field_name = some_pattern; ...; field_name = some_pattern }
```

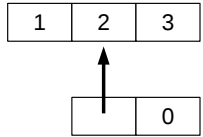
- (A record pattern may not mention all the record fields.)

In the machine

Program

```
let p = {
  x = 1;  y = 2; z = 3
}
let q = {
  b = p;
  s = 0
}
```

Machine



- A record is represented by a **heap-allocated block**.
- A record is represented exactly as a tuple.

Pitfalls: Typo in a field name

- Using type declaration, the compiler detects typo in a field identifier.

Typo in a field name I

```
type point2D = { x : int; y : int };;
# type point2D = { x : int; y : int; }
let p = { x = 42; z = 3 };;
# Characters 18-19:
  let p = { x = 42; z = 3 };;
                    ^
Error: Unbound record field z
```

Pitfalls: Missing field

- When constructing a record, all fields must be defined.

A field is missing I

```
type point2D = { x : int; y : int };;
# type point2D = { x : int; y : int; }
let oups = { x = 0 };;
# Characters 11-20:
  let oups = { x = 0 };;
             ^^^^^^^^^
Error: Some record fields are undefined: y
```

Pitfalls: Ill-typed field definition

- The value of each field must be compatible with the field type as declared by the record type definition.

A field is ill-typed I

```
type person = { name : string ; age : int };;
# type person = { name : string; age : int; }
let luke = { name = "Skywalker"; age = "26" };;
# Characters 39-43:
  let luke = { name = "Skywalker"; age = "26" };;
                                         ^^^^
Error: This expression has type string but an expression was
   expected of type
         int
```

Pitfalls: Shadowing a field name

- The compiler does its best to disambiguate the usage of labels, but sometimes the ambiguity cannot be fixed (and is probably not intended by the programmer).

Shared field names I

```
type a = { x : int; b : int; };;
# type a = { x : int; b : int; }
type b = { y : int; c : int; };;
# type b = { y : int; c : int; }
{ x = 0; b = 2 };;
# - : a = {x = 0; b = 2}
type t = { x : bool };;
# type t = { x : bool; }
type u = { x : int };;
# type u = { x : int; }
```

Shared field names II

```
{ x = true };;
# Characters 6-10:
  { x = true };;
        ^^^^
Error: This expression has type bool but an expression was expected
    of type
           int
```