Tuples and Records (Product types)

Ziyan Maraikar

June 29, 2016

Compound data types

- ★ We often need to group multiple data into a logical entity, e.g. the x and y coordinates of a point.
- ★ This is achieved using a *compound type*.
- ★ Ocaml provides *tuples* and *records*[1] for grouping logical entities.

[1] similar to a C struct

Creation and access

Ocaml provides two basic operations for each compound type.

- ★ An operation that *constructs* the compound type out of values.
- ★ Access to its component values using *patterns* to deconstruct (destructure) the compound type.

Lecture Outline

Tuple construction

A tuple is an ordered collection of values that can each be of a different type.

You construct a tuple by joining values together with a comma:

```
let day = (0, "Sunday")

let coordinate = (1., -1., 0)
```

By convention we enclose a tuple in parentheses in Ocaml (although it is not required.)

Tuple type

The type of a tuple is written with as the types of its constituents separated by a $*$.

```
let day = (0, "Sunday") ;;
val day : int * string = (0, "Sunday")

let coordinate = (1., -1., 0) ;;
val coordinate : float * float * int = (1., -1., 0)
```

The type `t * s` denotes the *Cartesian product* of the elements of type `t` and elements of type `s`.

Exercise

Write down the types of each of the following tuples.

- ★ `(1,(2,3))`
  The components of a tuple can be compound types.
- ★ `()`
  The empty tuple denotes a special type called `unit` that denotes the lack of a function result (similar to void in C.)

Lecture Outline

Extracting values using patterns

To extract values from a tuple we use *pattern matching*. We will use this Ocaml feature extensively.

```
let (x, y, z) =  (1., -1., 0)

let (name, age) = ("Silva", 30)
```

Note that the pattern must match the structure of the type. For tuples the arities must match.

```
let (x,y) = (1., -1., 0) ;;
Error: This expression has type 'a * 'b * 'c but an
expression was expected of type 'd * 'e
```

Tuple as a function result

Compound types such as tuples can be returned as the result of a function.

```
let scale (x:float) (y:float) (factor:float) :float =
  (x *. factor, y *. factor)
```

Tuples as function arguments

```
let euclidean_dist (p1: float*float) (p2: float*float) :
    float =
  let (x1, y1) = p1 in
  let (x2, y2) = p2 in
    ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.) ** 0.5
```

We can also use pattern matching when specifying function parameters

```
let euclidean_dist
((x1:float), (y1:float)) ((x2:float), (y2:float)) :float =
  ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.) ** 0.5
```

Exercise

Write functions to calculate the

- ★ dot product of two 3-dimensional vectors.
- ★ determinant of a 2×2 matrix.

Lecture Outline

Grouping data with Records

- ★ A record is a compound type, that groups together *fields* of various types (similar to tuples and C structs).
- ★ A record's *type definition* defines its field names and their respective types (unlike tuples.)

Record type definitions

```
type person = { fname: string; surname: string; age: int;
    married: bool }
```

The general form of a type definition is fields and their respective types separated by semicolons.

$$\texttt{type rectype = \{ } f_1 : t_1; \ldots; f_n : t_n \texttt{ \}}$$

Construction and field access

To construct a record we provide a value for each field.

```
let student = { fname="Lal"; surname="Silva"; age=20;
    married=false }
```

Fields of a record may be accessed using the dot.

```
let fullname = student.fname ^ " " ^ student.surname
```

Lecture Outline

Record deconstruction

Patterns can also be used to access a record's fields.

```
let student = { fname="Lal"; surname="Silva"; age=20;
    married=false }
let { fname=fn ; surname=sn; age=a; married=m } = student
```

This pattern can be written in a shorter form using variables which are the same as the field names,

```
let { fname; surname; age; married }  = student
```

This feature is called *field punning*.

Incomplete patterns

We can extract a subset of fields using ignoring fields we do not need.

```
let student = { fname="Lal"; surname="Silva"; age=20;
    married=false }
let { surname; age; }  = student
```

Exercise

1. Write a function that takes a `person` record and returns the full name in the form "*surname, fname*", e.g. "Silva, Lal".
2. Write a function that adds a given increment to a staff member's salary, given the following type definition.

```
type staff = { name:string; salary:int; married:bool }
```

Functional updates

- ★ Since records are immutable by default, we need to make a copy when updating a field.
- ★ *Functional update* syntax lets us avoid copying all the unchanged fields individually.

```
let increment_salary emp inc  =
  { emp with salary = emp.salary + inc }
```

The general functional update syntax is

{ recvar with $f_1 = v_1; \ldots; f_n = v_n$ }

Field name clashes

Given the following definitions, suppose we need a function to get the name of a staff member.

```
type staff = { name:string; salary:int; married:bool }
type student = { name:string; batch:string }
```

When two record types use the same field name, you should explicitly state the desired type.

```
let get_name ({ name; }:staff) = name ;;
val get_name: staff -> string = <fun>
```

Ocaml infers a record's type by matching field names. Here it cannot determine which of the types the pattern `{ name; }` should match.

```
let get_name { name; } = name ;;
val get_name: student -> string = <fun>
```