# Tuples and pattern matching

Ziyan Maraikar

July 20, 2014

# Table of Contents

# 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.

# Table of Contents

# Tuples

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.)

# Table of Contents

# 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 parameteres

```
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 $2 \times 2$ matrix.

# Match expressions

Match expressions permit matching an expression to a pattern containing *literals*, *variables*, or a combination of the two.

```
match e with
| p_1 -> r_1
| ...
| p_n -> r_n
```

- $p_1, p_2, \ldots$ are patterns which can match the value of the expression $e$. The types of $e$ and $p_i$s must all be the same.
- If pattern $p_i$ matches $e$, then result of `match` is the corresponding $r_i$ expression's value.

# Example

```
let factorial n =
  match n with
  | 0 -> 1
  | n -> n * fact (n-1)
```

Exercise: Write `fibonacci` using a match expression.

# Matching patterns with literals

- A pattern with literals $p^{v_1, v_2, \cdots}$, matches only if the values $v_1, v_2, \ldots$ are equal to the corresponding parts of the expression $e$.
- The order in which patterns are listed is important. The most specific pattern (containing the most literals) must appear first.

```
let factorial n =
  match n with
  | n -> n * fact (n-1)
  | 0 -> 1
```

The pattern `0` will never get a chance to match because the variable pattern `n` matches any value.

# Matching on tuples

We can provide a literal or variable for each element in a tuple match pattern.

```
let number_kind (real, imag) =
  match (real, imag) with
  | (_, 0) -> "real"
  | (0, _) -> "imaginary"
  | (_, _) -> "complex"
```

Note the special *wildcard* variable denoted by _ that can be used to match and discard a value.

# Matching patterns with variables

In the general case matching a pattern with variables takes the form,

$$
\texttt{match}\ e^{v_1, v_2, \dots}\ \texttt{with} \\
|\ p^{x_1, x_2, \dots}\ \texttt{->}\ r
$$

Matching a pattern containing variables $x_1, x_2, \dots$ to an expression containing values $v_1, v_2, \dots$ *desugars* to,

$$
\texttt{let}\ x_1 = v_1\ \texttt{in} \\
\dots \\
\texttt{let}\ x_n = v_n\ \texttt{in} \\
\quad r
$$

That is, the variables $x_1, x_2, \dots$ bind to the corresponding parts of the expression $e$, and can then be used in the result expression $r$

# Limitations of match

**Comparison of values cannot be done using patterns!**

```
let is_equal x y =
  match x with
  | y -> true
  | _ -> false
;;
Warning 11: this match case is unused.
```

We cannot check a condition such as `x=y` using match because the variable `y` in the match is NOT the same as the parameter `y`.

```
is_equal 0 0 ;;
- : bool = true
is_equal 0 1 ;;
- : bool = true
```

# Non-exhaustive match

When matching against a compound type, Ocaml ensures that all possibilities are covered.

```
let number_kind (real, imag) =
  match (real, imag) with
  | (_, 0) -> "real"
  | (0, _) -> "imaginary"
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: (1, 1)
```

Ensures that you do not accidentally ommit a possible match combination!