Records & Variants

Ziyan Maraikar

July 28, 2014

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.

```
type rectype = {  f1 : t1; . . . ; fn : tn }
```

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
```

Record patterns

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 the wildcard, ignore fields we do not need.

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

Note that unlike tuples, a single wildcard matches all fields not listed in the pattern.

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. content...

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

The general functional update syntax is

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

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>
```

Lecture Outline

Many forms in a single type

- ★ A variant[1] represents data that may take on multiple different forms, where each form is marked by an explicit tag.
- ★ For example, In a "paint" program we need represent various shapes like triangles, squares, and ellipses.
  It is useful to represent these shapes using a common type. (why?)
- ★ Variants provide a way to represent such complex data and organise the case-analysis on that information with pattern matching.

[1] corresponding to the union in C

Representing enumerations

- The simplest use of variants is to represent an *enumeration* — a collection of constants.
- We can define the colour pallete of a paint program using a variant. Each *constructor* is separated by a vertical bar.

```
type colour = Red | Blue | Green | Cyan | Magenta |
    Yellow
```

Constructor must be named beginning with an uppercase letter.

- We can now define variables of this type,

```
let brush = Red ;;
val brush : colour = Red
```

Ocaml infers the variable's type from the constructor used.

Pattern matching variants

Pattern matching on a variant is done by listing its constructors in a match expression.

```
(** Convert a colour to a (red, green, blue) triple **)
let colour_to_rgb (c:colour) =
  match c with
  | Red -> (255, 0, 0)
  | Green -> (0, 255, 0)
  | Blue -> (0, 0, 255)
  | Cyan -> (0, 255, 255)
  | Magenta -> (255, 0, 255)
  | Yellow -> (255, 255, 0)
```

Note the correspondence between the match cases and the type definition.

Exercise

- ★ Add the constructors `Black` and `White` to the `colour` type.
- ★ What happens if you do not add them to the match expression in `colour_to_rgb`?
- ★ Using a wildcard to match variant constructors is possible, but bad programming practice!

Constructors with parameters

A constructor for a variant may have a list of fields associated with it.
A variant representing graphic elements in our paint program can be defined as,

```
type element =
(* Circle's centre coordinates and radius *)
| Circle of float * float * float
(* Line's end-point coordinates *)
| Line of float * float * float * float
```

We construct an element by giving its constructor the necessary field values in tuple-like syntax,

```
let c1 = Circle (1.0, 3.0, 2.0)
```

This is confusing since the order of the constructor's fields is not clear.

Exercise

- ★ Add the constructor `Text`. It should contain the string of text, font size and position.
- ★ Add a `colour` field to each constructor in `element`.

Pattern matching constructors with fields

```
(** Move an element a given x and y offset **)
let translate (e:element) ((dx,dy):float*float) =
  match e with
  | Circle(x, y, r)
    -> Circle (x+.dx, y+.dy, r)
  | Line(x1, y1, x2, y2)
    -> Line(x1+.dx, y1+.dy, x2+.dx, y2+.dy)
```

Note the similarity to tuple patterns (although these are not tuples.)

Combining records and variants

We can make element constructors more readable by introducing a record for representing point coordinates[2],

```
type point2d = { x:float; y:float }
type element =
  | Circle of point2d * float (* centre coordinates and radius *)
  | Line of point2d * point2d (* end-point coordinates *)
```

We must now put our coordinates within a `point2d` record,

```
let c1 = Circle ({x=1.0; y=3.0}, 2.0)
```

[2] We could have also used tuples instead

Pattern matching on complex types

```
(** Move an element a given x and y offset **)
let translate e (dx,dy) =
  let move {x; y} =
    {x=x+.dx; y=y+.dy} in
  match e with
  | Circle(c, r) -> Circle (move c, r)
  | Line(p1, p2) -> Line(move p1, move p2)
```

Exercise

- ★ Use a tuple instead of a record to store (x, y) coordinates in `element` constructors.
- ★ Define a function that rotates elements a given angle around the origin.

Variant syntax

The general syntax for defining variants is

```
type vname =
| Con1
...
| Con1 of t1 * ... * tn
```

Although the field definition syntax resembles tuples, fields are not stored as tuples.