Discriminated Unions (Sum type)

Ziyan Maraikar

July 5, 2016

Lecture Outline

1. Discriminated Unions

2. Pattern matching

3. Constructors with parameters

Representing shapes as types

Suppose we need to represent shapes in a drawing program, like

- lines
- polygons
- circles and ellipses.

Is useful to represent these shapes using a common type?

Representing shapes as types

Problem: each shapes is specified differently

- line is specified by two end-points
- a polygon is specified by n vertices
- a circle is specified by a centre and radius

How would we model this in an OO language like Java?

Many forms in a single type

- A *discriminated union*[1] type can represent such heterogeneous forms of data.
- Each form is marked by a named *constructor* (or *tag*), which is used in case-analysis with *pattern matching*.

[1] roughly corresponds to Java enums

Representing enumerations

The simplest use of discriminated unions is to represent *symbolic data.*

Example: To define the colour palette of a paint program we may use a constructor for each colour name.

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

In Ocaml constructors must begin with an upper-case 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.

Lecture Outline

1. Discriminated Unions
2. Pattern matching
3. Constructors with parameters

Deconstructing discriminated unions

To Deconstruct a discriminated union, we pattern match on its constructors.

```
(** 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 cases in the match expression and the definition of type `colour`

MATCH EXPRESSIONS

Match expressions permit matching an expression to a *pattern* containing literals and variables.

```
match e with
| p1 -> r1
| ...
| pn -> rn
```

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

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 discriminated union constructors is possible, but bad programming practice!

Lecture Outline

1. Discriminated Unions
2. Pattern matching
3. Constructors with parameters

Constructors with parameters

A constructor for a discriminated union may contain parameters.

Example: A discriminated union representing graphic shapes in our paint program can be defined as,

```
type shape =
(*  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 shape by giving its constructor the necessary field values in tuple-like syntax,

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

General discriminated union definition

The general syntax for defining discriminated unions is

```
type du_name =
| C1
...
| Cn of t1 * ... * tn
```

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

Exercise

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

Matching on constructors with arguments

```
(** Move an shape a given x and y offset **)
let translate (s:shape) ((dx,dy):float*float) =
  match s 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.)

Semantics of match expressions

In general the expression

$$
\begin{array}{l}
\texttt{match}\ C_i(v)\ \texttt{with} \\
|C_1(x_1) \rightarrow e_1[x_1] \\
\ldots \\
|C_1(x_n) \rightarrow e_n[x_n]
\end{array}
$$

evaluates to

$$e_i[x_i/v]$$

Types in match expressions

$$
\begin{array}{l}
\texttt{match}\ C_i(v)\ \texttt{with} \\
|\,C_1(x_1) \to e_1[x_1] \\
\ldots \\
|\,C_1(x_n) \to e_n[x_n]
\end{array}
$$

What can be said about the types of C_i, $C_1, \ldots, C_n$ and $e_1[x_1], \ldots, e_n[x_n]$?

Exercise

Evaluate the expression

```
let c = Circle (0.0, 0.0, 1.0) in
  translate c (1.0,1.0)

let translate (s:shape) ((dx,dy):float*float) =
  match s 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)
```

Combining records and discriminated unions

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

```
type point2d = { x:float; y:float }
type shape =
  | 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

Matching on complex types

```
(** Move an shape 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 `shape` constructors.
- Define a function that rotates shapes a given angle around the origin.

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 patterns appear in is important. The most specific pattern (containing the most literals) must appear first, or it may never be matched.

```
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` already matches any value.

Matching tuple patterns

We can provide a literal or variable for each shape 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 not used in the result expression.

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.

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

The variable `y` in the match is not the same as the parameter `y`. So the pattern `y` matches any values regardless of `x`.

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)
```

This check ensures you do not accidentally omit a possible match combination!