Records

Ziyan Maraikar

July 25, 2014

Records

- A record is a compound type, where each component is identified by a field name.

Records

- ★ A record is a compound type, where each component is identified by a field name.
- ★ The type of a record must be defined before use.

Records

- A record is a compound type, where each component is identified by a field name.
- The type of a record must be defined before use.
- Similar to structs in C.

Type definitions

We introduce a user-defined type in Ocaml using a *type definition*.

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

Type definitions

We introduce a user-defined type in Ocaml using a *type definition*.

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

The general form of a type definition is

```
type typename = { field1 : type1; ...; fieldn : typen }
```

Construction and field access

To construct a record we provide the corresponding field values.

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

Fields are separated by semicolons.

Construction and field access

To construct a record we provide the corresponding field values.

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

Fields are separated by semicolons.
Fields of a record are accessed using the dot.

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

Record patterns

Pattern matching can also be used to access fields.

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

Record patterns

Pattern matching can also be used to access fields.

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

If we use the record's field names as variables we can also write the pattern match as,

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

This is called *field punning*.

Matching on incomplete patterns

We can match a subset of record fields using the wildcard.

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

Matching on incomplete patterns

We can match a subset of record fields using the wildcard.

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

Exercise: Write a function that takes a `person` record and returns the full name in the form `"surname, fname"`, e.g. "Silva, Lal".

Functional updates

Since records are immutable by default, we need to make a copy when updating a field.
Exercise: write a function that adds a given increment to a staff member's salary.

Functional updates

Since records are immutable by default, we need to make a copy when updating a field.
Exercise: write a function that adds a given increment to a staff member's salary. Instead of copying fields individually, we can use update a field more concisely using *functional update* syntax.

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

Functional updates

Since records are immutable by default, we need to make a copy when updating a field.
Exercise: write a function that adds a given increment to a staff member's salary. Instead of copying fields individually, we can use update a field more concisely using *functional update* syntax.

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

The general syntax is

```
{ record with field1 = val1; ...; fieldn : valn }
```

Field name clashes

Ocaml inferes the type of a record by matching field names. This causes problems if two record types use the same field name.

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

Field name clashes

Ocaml inferes the type of a record by matching field names. This causes problems if two record types use the same field name.

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

Suppose we need a function to get the name of a staff member.

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

Field name clashes

Ocaml inferes the type of a record by matching field names. This causes problems if two record types use the same field name.

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

Suppose we need a function to get the name of a staff member.

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

To avoid this problem we need be explicit about the parameter type

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