OCaml III: Unions and Products

CAS CS 320: Principles of Programming Languages

January 30, 2024

Outline

Administrivia

- Homework 1 is due on *Thursday by 11:59PM*. This assignment is graded.
- Discussions will be held this week to practice the material we saw last week.
- Please talk to me ASAP if you require accommodations for this course.

Today

Examine the notions of unions and products and see how these can be used to model data in complex programs.

Start to look more carefully at patterns and pattern matching.

Keywords

- variants
- unions
- pattern matching
- variable patterns
- wildcards
- tuples
- records
- products
- fields
- accessors
- record updates

Reap Problem

What is the value of v?

```
let v =
  let rec f x = g (x + 1)
  and g x = 1 + h (x - 1)
  and h x = if x < 0 then 0 else f (x - 2)
  in f 8
```

1. 0
2. 6
3. 8
4. No value

Solution

```
f 8          (* ==> *);;
g 9          (* ==> *);;
1 + h 8      (* ==> *);;
1 + f 6      (* ==> *);;
2 + f 4      (* ==> *);;
3 + f 2      (* ==> *);;
4 + f 0      (* ==> *);;
5 + f (-2) (* ==> *);;
6 + h (-2) (* ==> *);;
6            (* fin *);;
```

Outline

Début

Variants (Unions)

Tuples and Records (Products)

Fin

Enum-Like Variants

A variant is a type for values of a fixed set of possiblities. Simple variants are similiar to `enum`'s in Java.

Example.

```
type myunit = Unit
type mybool = True | False
```

- `myunit` is called a variant. `Unit` is called a tag or a constructor.
- user-defined variants are lowercase, constructors are Uppercase.

Enum-Like Variants

A variant is a type for values of a fixed set of possiblities. Simple variants are similiar to `enum`'s in Java.

Example.

```
type myunit = Unit
type mybool = True | False
```

- `myunit` is called a variant. `Unit` is called a tag or a constructor.
- user-defined variants are lowercase, constructors are Uppercase.

Enum-Like Variants

A variant is a type for values of a fixed set of possiblities. Simple variants are similiar to `enum`'s in Java.

Example.

```
type myunit = Unit
type mybool = True | False
```

- `myunit` is called a variant. `Unit` is called a tag or a constructor.
- user-defined variants are lowercase, constructors are Uppercase.

Enum-Like Variants

A variant is a type for values of a fixed set of possiblities. Simple variants are similiar to `enum`'s in Java.

Example.

```
type myunit = Unit
type mybool = True | False
```

- `myunit` is called a variant. `Unit` is called a tag or a constructor.
- user-defined variants are lowercase, constructors are Uppercase.

Enum-Like Variants: Syntax and Semantics

Syntax.

```
type t = Const_1 | Const_2 | ... | Const_n
```

Dynamics Semantics. *There is none.* This is the point. `Const_i` is a value by definition.

Static Semantics. If `t` is defined as

```
type t = ... | C | ...
```

then `C : t`. That is, every constructor of a variant `t` is of type `t`.

Enum-Like Variants: Syntax and Semantics

Syntax.

```
type t = Const_1 | Const_2 | ... | Const_n
```

Dynamics Semantics. *There is none.* This is the point. `Const_i` is a value by definition.

Static Semantics. If `t` is defined as

```
type t = ... | C | ...
```

then `C : t`. That is, every constructor of a variant `t` is of type `t`.

Enum-Like Variants: Syntax and Semantics

Syntax.

```
type t = Const_1 | Const_2 | ... | Const_n
```

Dynamics Semantics. *There is none.* This is the point. `Const_i` is a value by definition.

Static Semantics. If `t` is defined as

```
type t = ... | C | ...
```

then `C : t`. That is, every constructor of a variant `t` is of type `t`.

Pattern Matching with Variants

Working with a variant means writing "what you want" for each constructor. In OCaml, this means using a match expression.

Example:

```
type my_bool = True | False

let my_not b =
  match b with
  | True -> False
  | False -> True

let _ = assert (my_not True = False)
```

This is sometimes called destructing.

Pattern Matching with Variants

Working with a variant means writing "what you want" for each constructor. In OCaml, this means using a match expression.

Example:

```
type my_bool = True | False

let my_not b =
  match b with
  | True -> False
  | False -> True

let _ = assert (my_not True = False)
```

This is sometimes called destructing.

Pattern Matching with Variants

Working with a variant means writing “what you want” for each constructor. In OCaml, this means using a match expression.

Example:

```
type my_bool = True | False

let my_not b =
  match b with
  | True -> False
  | False -> True

let _ = assert (my_not True = False)
```

This is sometimes called destructing.

Patterns

Patterns are typed templates for how a piece of data should look.

Terminology. In the expression below, we match on an expression e and the value of e matches with the pattern p.

```
match e with
| p -> o
...
```

Important. A pattern is *not* the same thing as a value or an expression.

Patterns

Patterns are typed templates for how a piece of data should look.

Terminology. In the expression below, we match on an expression `e` and the value of `e` matches with the pattern `p`.

```
match e with
| p -> o
...
```

Important. A pattern is *not* the same thing as a value or an expression.

Patterns

Patterns are typed templates for how a piece of data should look.

Terminology. In the expression below, we match on an expression `e` and the value of `e` matches with the pattern `p`.

```
match e with
| p -> o
...
```

Important. A pattern is *not* the same thing as a value or an expression.

Matching with a Pattern

For every type we have to specify:

- What are the patterns of that type (i.e., what kind of patterns can match its values)?
- How do values of a given type match with patterns of a given type?

There are general patterns which work for any type:

- Constants
- Variables
- Wildcards

Note. We can pattern match on *any* type in OCaml.

Matching with a Pattern

For every type we have to specify:

- What are the patterns of that type (i.e., what kind of patterns can match its values)?
- How do values of a given type match with patterns of a given type?

There are general patterns which work for any type:

- Constants
- Variables
- Wildcards

Note. We can pattern match on *any* type in OCaml.

Matching with a Pattern

For every type we have to specify:

- What are the patterns of that type (i.e., what kind of patterns can match its values)?
- How do values of a given type match with patterns of a given type?

There are general patterns which work for any type:

- Constants
- Variables
- Wildcards

Note. We can pattern match on *any* type in OCaml.

Matching with a Pattern

For every type we have to specify:

- What are the patterns of that type (i.e., what kind of patterns can match its values)?
- How do values of a given type match with patterns of a given type?

There are general patterns which work for any type:

- Constants
- Variables
- Wildcards

Note. We can pattern match on *any* type in OCaml.

General Patterns: Constants

Any value (also called a constant) is a valid pattern. A value matches with a constant pattern if it is equal.

Example:

```
let is_seven_or_eleven n =
  match n with
  | 7 -> true
  | 11 -> true
  | _ -> false
```

General Patterns: Constants

Any value (also called a constant) is a valid pattern. A value matches with a constant pattern if it is equal.

Example:

```
let is_seven_or_eleven n =
  match n with
  | 7 -> true
  | 11 -> true
  | _ -> false
```

General Patterns: Variables

A variable can be used as named default pattern. A variable matches any value.

Example.

```
let rec fib n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | k -> fib (k - 1) + fib (k - 2)
```

General Patterns: Variables

A variable can be used as named default pattern. A variable matches any value.

Example.

```
let rec fib n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | k -> fib (k - 1) + fib (k - 2)
```

General Patterns: Wildcards

An wildcard (written as an underscore _) can be used as an unnamed default pattern. A wildcard matches any value.

Example:

```
type day = M | T | W | Th | F | S | Su

let is_weekend d =
  match d with
  | S -> true
  | Su -> true
  | _ -> false
```

General Patterns: Wildcards

An wildcard (written as an underscore _) can be used as an unnamed default pattern. A wildcard matches any value.

Example:

```
type day = M | T | W | Th | F | S | Su

let is_weekend d =
  match d with
  | S -> true
  | Su -> true
  | _ -> false
```

Match Expressions: General Syntax

Given the expressions e, e_1, e_2, …, e_k and patterns p_1,…, p_k, we can write the following expression.

General Match Expression:

```
match e with
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

Match Expressions: Dynamic Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If e evaluates to v and
- v matches with p_i and
- p_i is the first pattern v matches with, and
- e_i evaluates to w then
- m evaluates to w.

Match Expressions: Dynamic Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If e evaluates to v and
- v matches with p_i and
- p_i is the first pattern v matches with, and
- e_i evaluates to w then
- m evaluates to w.

Match Expressions: Dynamic Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If e evaluates to v and
- v matches with p_i and
- p_i is the first pattern v matches with, and
- e_i evaluates to w then
- m evaluates to w.

Match Expressions: Dynamic Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If `e` evaluates to `v` and
- `v` matches with `p_i` and
- `p_i` is the first pattern `v` matches with, and
- `e_i` evaluates to `w` then
- `m` evaluates to `w`.

Match Expressions: Dynamic Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If e evaluates to v and
- v matches with p_i and
- p_i is the first pattern v matches with, and
- e_i evaluates to w then
- m evaluates to w.

Match Expressions: Static Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If `e` is of type `t` and
- all `p_i` are of type `t` and
- all `e_i` are of type `o` then
- `m` is of type `o`.

Match Expressions: Static Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If `e` is of type `t` and
- all `p_i` are of type `t` and
- all `e_i` are of type `o` then
- `m` is of type `o`.

Match Expressions: Static Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If `e` is of type `t` and
- all `p_i` are of type `t` and
- all `e_i` are of type `o` then
- `m` is of type `o`.

Match Expressions: Static Semantics

General Match Expression:

```
match e with (* referred to as `m` below *)
| p_1 -> e_1
| p_2 -> e_2
...
| p_k -> e_k
```

- If `e` is of type `t` and
- all `p_i` are of type `t` and
- all `e_i` are of type `o` then
- `m` is of type `o`.

Examples

What does `e` evaluate to?

```
let e = match true with
  | true -> 1
  | false -> 0
```

1. `0 : int`
2. `1 : int`
3. No value
4. Syntax error

Examples

What does `e` evaluate to?

```
let f x = x mod 3
let e = match f 13 with
  0 -> 1
  1 -> 0
  _ -> assert false
```

1. `0 : int`
2. `1 : int`
3. No value
4. Syntax error

Examples

What does `e` evaluate to?

```
let x = 10
let e = match 11 with
  | x ->
    (match x with
     | 10 -> 0
     | 11 -> 1
     | i -> i)
  | _ -> assert false
```

1. `0 : int`
2. `1 : int`
3. No value
4. Syntax error

Practical Example: Mime Types

Example:

```
module Type = struct
type t =
  | Text
  | Image
  | Audio
  | Video
  | Application
  | Message
  | Multipart
```

Attachments in an email are labeled with mime types which describe what kind of file the attachment it is.

If you want to build an representation of mime types in OCaml, you can use a simple variant.

Question

Implement the function `after_day` of type `day -> int -> day` which, given a day `d` and integer `i` returns the day which is `i` days after `d`.

Starter:

```
type day = M | T | W | Th | F | S | Su

let after_day (d : day) (i : int) : day =
  assert false (* TODO *)

let _ = assert (after_day W 17 = F)
let _ = assert (after_day W (-1) = T)
```

Data-Carrying Variants

Constructors of a variant can also hold data of other types.

Example:

```
type int_or_string =
  | Int of int
  | String of string

let two_in_box : int_or_string = Int 2
```

We will talk a lot more about this next week when we cover algebraic data types.

Data-Carrying Variants

Constructors of a variant can also hold data of other types.

Example:

```
type int_or_string =
  | Int of int
  | String of string

let two_in_box : int_or_string = Int 2
```

We will talk a lot more about this next week when we cover algebraic data types.

Data-Carrying Variants

Constructors of a variant can also hold data of other types.

Example:

```
type int_or_string =
  | Int of int
  | String of string

let two_in_box : int_or_string = Int 2
```

We will talk a lot more about this next week when we cover algebraic data types.

Patterns for Data-Carrying Variants

Example:

```
let int_of_int_or_string x =
  match x with
  | Int n -> n
  | String s -> int_of_string s

let _ = assert (int_of_int_or_string (Int 2) = 2)
let _ = assert (int_of_int_or_string (String "2") = 2)
```

New Pattern Rules.

- If `C` is a constructor for `t` which carries something of type `s`, and `p` is a pattern for `s`, then `C p` is a pattern of type `t`.
- `C p` matches `C v` if `v` matches `p`.

Patterns for Data-Carrying Variants

Example:

```ocaml
let int_of_int_or_string x =
  match x with
  | Int n -> n
  | String s -> int_of_string s

let _ = assert (int_of_int_or_string (Int 2) = 2)
let _ = assert (int_of_int_or_string (String "2") = 2)
```

New Pattern Rules.

- If `C` is a constructor for `t` which carries something of type `s`, and `p` is a pattern for `s`, then `C p` is a pattern of type `t`.
- `C p` matches `C v` if `v` matches `p`.

What does this have to do with unions?

The union of two collections A and B is the collection of objects in A or B.

It's not hard to envisage this type as the "union" of the types `string` and `int`.

```
type int_or_string =
  | Int of int
  | String of string

let two_int : int_or_string = Int 2
let two_str : int_or_string = String "two"
```

(If you squint.)

What does this have to do with unions?

The union of two collections A and B is the collection of objects in A or B.

It's not hard to envisage this type as the “union” of the types `string` and `int`.

```
type int_or_string =
  | Int of int
  | String of string

let two_int : int_or_string = Int 2
let two_str : int_or_string = String "two"
```

(If you squint.)

Outline

Tuples should be familiar

Tuples are ordered fixed-length collections of unnamed data.

Example:

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

They are useful for *small* “packets” of data whose individual pieces are not ambiguous.

Tuples should be familiar

Tuples are ordered fixed-length collections of unnamed data.

Example:

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

They are useful for *small* “packets” of data whose individual pieces are not ambiguous.

Tuples: Syntax

Given expressions `e_1`, `e_2`,…, `e_k`, we have the expression

```
(e_1, e_2,..., e_k)
```

Given types `t_1`, `t_2`,…, `t_k`, we have the type

```
t_1 * t_2 * ... * t_k
```

Tuples: Dynamic Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If `e_1` evaluates to `v_1` and
- `e_2` evaluates to `v_2` and
- …
- `e_k` evaluates to `v_k` then
- `e` evaluates to `(v_1, v_2,..., v_k)`

Tuples: Dynamic Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 evaluates to v_1 and
- e_2 evaluates to v_2 and
- …
- e_k evaluates to v_k then
- e evaluates to (v_1, v_2,..., v_k)

Tuples: Dynamic Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 evaluates to v_1 and
- e_2 evaluates to v_2 and
- …
- e_k evaluates to v_k then
- e evaluates to (v_1, v_2,..., v_k)

Tuples: Dynamic Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 evaluates to v_1 and
- e_2 evaluates to v_2 and
- …
- e_k evaluates to v_k then
- e evaluates to (v_1, v_2,..., v_k)

Tuples: Static Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 is of type t_1 and
- e_2 is of type t_2 and
- …
- e_k is of type t_k then
- e is of type t_1 * t_2 * ... * t_k

Tuples: Static Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 is of type t_1 and
- e_2 is of type t_2 and
- …
- e_k is of type t_k then
- e is of type t_1 * t_2 * ... * t_k

Tuples: Static Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If e_1 is of type t_1 and
- e_2 is of type t_2 and
- …
- e_k is of type t_k then
- e is of type t_1 * t_2 * ... * t_k

Tuples: Static Semantics

General Tuple Syntax:

```
(e_1, e_2,..., e_k) (* refered to as `e` below *)
```

- If `e_1` is of type `t_1` and
- `e_2` is of type `t_2` and
- …
- `e_k` is of type `t_k` then
- `e` is of type `t_1 * t_2 * ... * t_k`

Tuples: Pattern Matching

General Tuple Pattern Syntax:

```
(p_1, p_2,..., p_k) (* refered to as `p` below *)
```

- If p_1 is of type t_1 and v_1 matches p_1
- p_2 is of type t_2 and v_2 matches p_2
- …
- p_k is of type t_k and v_k matches p_k
- p is of type t_1 * t_2 * ... * t_k and
- (v_1, v_2,..., v_k) matches p

Note. There are no accessors for tuples.

Tuples: Pattern Matching

General Tuple Pattern Syntax:

```
(p_1, p_2,..., p_k) (* refered to as `p` below *)
```

- If `p_1` is of type `t_1` and `v_1` matches `p_1`
- `p_2` is of type `t_2` and `v_2` matches `p_2`
- …
- `p_k` is of type `t_k` and `v_k` matches `p_k`
- `p` is of type `t_1 * t_2 * ... * t_k` and
- `(v_1, v_2,..., v_k)` matches `p`

Note. There are no accessors for tuples.

Tuples: Pattern Matching

General Tuple Pattern Syntax:

```
(p_1, p_2,..., p_k) (* refered to as `p` below *)
```

- If p_1 is of type t_1 and v_1 matches p_1
- p_2 is of type t_2 and v_2 matches p_2
- …
- p_k is of type t_k and v_k matches p_k
- p is of type t_1 * t_2 * ... * t_k and
- (v_1, v_2,..., v_k) matches p

Note. There are no accessors for tuples.

Tuples: Pattern Matching

General Tuple Pattern Syntax:

```
(p_1, p_2,..., p_k) (* refered to as `p` below *)
```

- If `p_1` is of type `t_1` and `v_1` matches `p_1`
- `p_2` is of type `t_2` and `v_2` matches `p_2`
- …
- `p_k` is of type `t_k` and `v_k` matches `p_k`
- `p` is of type `t_1 * t_2 * ... * t_k` and
- `(v_1, v_2,..., v_k)` matches `p`

Note. There are no accessors for tuples.

Example

Euclidean Distance:

```
let dist p q =
  match p with
    | (x1, y1) ->
      match q with
      | (x2, y2) ->
        let x_diff = x1 -. x2 in
        let y_diff = y1 -. y2 in
        sqrt (x_diff *. x_diff +. y_diff *. y_diff)
```

In reality, this is not how you'd write this function, let's demo the correct way.

Pattern Matching in Function Arguments

We can use patterns as arguments to functions instead of just names.

Example.

```
let dist (x1, y1) (x2, y2) =
  let xd = x1 -. x2 in
  let yd = y1 -. y2 in
  sqrt (xd *. xd +. yd *. yd)
```

Records

Records are unordered fixed-length collections of named data.

Example:

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }
```

They are useful for organizing larger collections of related data.

Records

Records are unordered fixed-length collections of named data.

Example:

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }
```

They are useful for organizing larger collections of related data.

Records

Records are unordered fixed-length collections of named data.

Example:

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }
```

They are useful for organizing larger collections of related data.

Records: Syntax

- Given fields `f_1`, `f_2`,…, `f_k` and
- expressions `e_1`, `e_2`,…, `e_k` and
- types `t_1`, `t_2`,…, `t_k` we have the expression

```
{ f_1 = e_1 ;
  f_2 = e_2 ;
  ...
  f_k = e_k ;
}
```

and the type

```
{ f_1 : t_1 ;
  f_2 : t_2 ;
  ...
  f_k : t_k ;
}
```

Records: Semantics, Patterns

I won’t dwell on this one. Its similar to the case of tuples.

See the textbook for more details.

Accessors

Records *do* have accessors. We can use dot-notation.

Example:

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

It's less common to pattern match on records because they can often be quite large (though it is still possible).

Accessors

Records *do* have accessors. We can use dot-notation.

Example:

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

It's less common to pattern match on records because they can often be quite large (though it is still possible).

Accessors

Records *do* have accessors. We can use dot-notation.

Example:

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

It's less common to pattern match on records because they can often be quite large (though it is still possible).

Practical Example

Managing Users:

```
type userType = Beginner | Editor | Visitor | Moderator

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
  ty : userType ;
}

let init_user name email vis : user = {
  name = name ;
  email = email ;
  num_posts = 0;
  ty = if vis then Visitor else Beginner ;
}
```

Record Updates

Since we often only care about changing a small portion of a record, there is record-update syntax for doing this.

Example:

```
let update_name u new_name new_email : user =
  { u with name = new_name ; email = new_email }

let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

- We read `{ u with ... }` as "the record `u` but with … changed".
- Remember, we are in the functional paradigm. We can't actually update records, we always return a new record.

Record Updates

Since we often only care about changing a small portion of a record, there is record-update syntax for doing this.

Example:

```
let update_name u new_name new_email : user =
  { u with name = new_name ; email = new_email }

let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

- We read { u with ... } as “the record u but with … changed”.
- Remember, we are in the functional paradigm. We can’t actually update records, we always return a new record.

Record Updates

Since we often only care about changing a small portion of a record, there is record-update syntax for doing this.

Example:

```
let update_name u new_name new_email : user =
  { u with name = new_name ; email = new_email }

let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

- We read `{ u with ... }` as "the record `u` but with … changed".
- Remember, we are in the functional paradigm. We can't actually update records, we always return a new record.

Record Updates

Since we often only care about changing a small portion of a record, there is record-update syntax for doing this.

Example:

```
let update_name u new_name new_email : user =
  { u with name = new_name ; email = new_email }

let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

- We read `{ u with ... }` as "the record `u` but with … changed".
- Remember, we are in the functional paradigm. We can't actually update records, we always return a new record.

What does this have to do with products?

The product of two collections A and B is the set of ordered pairs (a, b) where a comes from A and b comes from B.

The relation with tuples is obvious.

```
type int_times_string = int * string
```

What does this have to do with products?

The product of two collections A and B is the set of ordered pairs (a, b) where a comes from A and b comes from B.

The relation with tuples is obvious.

```
type int_times_string = int * string
```

Question

Write a function `to_polar` of type `point -> p_coord` which converts a point in Cartesian coordinates to one in polar coordinates. The functions `atan` and `sqrt` may be useful here.

Starter:

```
type point = { x : float ; y : float }
type p_coord = { d : float ; ang : float }

let to_polar (p : point) = (* TODO *)
```

Outline

Summary

Variants can be used to create a type whose values may be one of many kinds of things.

Patterns and match statements are used to work with those multiple kinds of things.

Tuples and records can be used to create types whose values contain many things in a single package. This is useful for organizing data.