Abstraction

- Programming languages have internal mechanisms for constructing and extending their operations and data types.
  - Functional abstraction (functions): extends the basic operations of the language.
  - Type abstraction (polymorphism): extends the application of operations to values of different kinds.
  - Data abstraction (composite types): extends the available values (and types) for expressing computations.

- Primitive types: integers, reals, booleans, strings, characters, etc.
- Composite types: pairs (tuples), records, variants, lists, algebraic data types (ADTs).

The tuple

- In the **Curry-Howard** correspondence, the pair type represents the **conjunction**.
- It corresponds to the definition of the Cartesian product of two (or more) sets in the correspondence between types and set.
- It defined the type of values **that have two** (or more) **parts** of (potentially) different types.

$$A \times B$$

- The primitive tuple type allows for a finite number of components, with each component being of a type.
- The tuple type enable the combination of values in a heterogeneous way, as opposed to traditional data structures, which are homogeneous collections.

The tuple

```
(1,2)
```

[4] ✓ 0.0s

```
- : int * int = (1, 2)
```

A tuple is an immutable value!

$$\frac{A \quad B}{A \wedge B}$$

```
("The answer is", 42)
```

[5] ✓ 0.0s

```
- : string * int = ("The answer is", 42)
```

```
("FullName",("John","Doe"))
```

[9] ✓ 0.0s

```
- : string * (string * string) = ("FullName", ("John", "Doe"))
```

The tuple

```
let pairup = fun x y -> (x,y);;

pairup "Hello" (pairup 1 2)
```

[11] ✓ 0.0s

```
val pairup : 'a -> 'b -> 'a * 'b = <fun>

- : string * (int * int) = ("Hello", (1, 2))
```

$$\frac{A \quad B}{A \wedge B}$$

```
let rotate_point (x, y) theta =
  let cos_theta = cos theta in
  let sin_theta = sin theta in
  let new_x = x *. cos_theta -. y *. sin_theta in
  let new_y = x *. sin_theta +. y *. cos_theta in
  (new_x, new_y)
```

[7] ✓ 0.0s

```
val rotate_point : float * float -> float -> float * float = <fun>
```

```
let rotated_point = rotate_point (3.0, 4.0) (Float.pi /. 4.0)
```

[8] ✓ 0.0s

```
val rotated_point : float * float = (-0.707106781186547, 4.94974746830583268)
```

Elimination of a tuple

```
let p = (1,2) in fst p + snd p
```

[15] ✓ 0.0s

```
- : int = 3
```

```
let (x,y) = (1,2) in x + y
```

[16] ✓ 0.0s

```
- : int = 3
```

```
(fun (x,y) -> x + y ) (1,2)
```

[17] ✓ 0.0s

```
- : int = 3
```

$$\frac{A \wedge B}{A}$$

$$\frac{A \wedge B}{B}$$

Type definition by name

- Composite types become “reusable” when associated with a new name.
- With type inference, this aspect is only particularly important in large applications and/or when we want to have “opaque” types in modules.
- We will later return to the definition of modules and module signatures.
- They are also important for defining records.

```
type coordinates = float * float
```

[33] ✓ 0.0s

```
type coordinates = float * float
```

```
module type PointsSig = sig
  type coordinates
  val create_point : float -> float -> coordinates
  val rotate_point : coordinates -> float -> coordinates
end

module PointsImpl = struct
  type coordinates = float * float

  let rotate_point (point: coordinates) theta =
    let cos_theta = cos theta in
    let sin_theta = sin theta in
    let new_x = (fst point) *. cos_theta -. (snd point) *. sin_theta in
    let new_y = (fst point) *. sin_theta +. (snd point) *. cos_theta in
    (new_x, new_y)

end
```

[34] ✓ 0.0s

Records

- Records are composite types where the components are accessed by name.

By default, record fields are immutable!

```
type person = {name: string; age: int}
[33] ✓ 0.0s
··· type person = { name : string; age : int; }
```

```
let p = {name = "John"; age = 42}
[35] ✓ 0.0s
··· val p : person = {name = "John"; age = 42}
```

```
"The person's name is " ^ p.name ^ " and age is " ^ string_of_int p.age
[37] ✓ 0.0s
··· - : string = "The person's name is John and age is 42"
```

```
fun {name;age} -> "The person's name is " ^ name ^ " and age is " ^ string_of_int age
[13] ✓ 0.0s
··· - : person -> string = <fun>
```

Functions with multiple parameters and multiple results

- The traditional way of calling a function, where all parameters are given at once, is done with tuples in OCaml.
- Tuples also allow returning multiple values simultaneously.
- The ability to call a function partially is called Currying.

```
let rotate_point (p, theta) =
  let (x,y) = p in
  let cos_theta = cos theta in
  let sin_theta = sin theta in
  let new_x = x *. cos_theta -. y *. sin_theta in
  let new_y = x *. sin_theta +. y *. cos_theta in
  (new_x, new_y)
```

[19] ✓ 0.0s

```
val rotate_point : (float * float) * float -> float * float = <fun>
```

```
let point = (3.0, 4.0) in
let rotated_point = rotate_point (point, (Float.pi /. 4.0))
```

[17] ✓ 0.0s

```
val rotated_point : float * float = (-0.707106781186547, 4.94974746830583268)
```

Sum types (or Variants)

- In the Curry-Howard correspondence, the sum type represents the **disjunction**.
- It corresponds to the definition of the (tagged) union of two (or more) sets in the correspondence between types and sets.
- It defines the type of values that **have one of two** (or more) **parts** of potentially different types.

$$A + B$$

- The primitive sum type allows a finite number of alternatives, where each alternative can be of any type.
- The sum type corresponds to combining different values in a heterogeneous way.

Sum type

- A sum type takes several possible alternatives for its values. It is a form of **ad-hoc polymorphism**.

```
type species = Dog | Cat | Bird | Fish
type pet = { name: string; species: species; age: int }
```

[20] ✓ 0.0s

```
type species = Dog | Cat | Bird | Fish

type pet = { name : string; species : species; age : int; }
```

```
let p = {name = "Fido"; species = Dog; age = 3}
```

[ ]

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

Sum type: data types

```
type point = float * float

type figure =
  | Circle of point * float
  | Rectangle of point * point
  | Triangle of point * point * point
```

[41] ✓ 0.0s

```
type point = float * float

type figure =
    Circle of point * float
  | Rectangle of point * point
  | Triangle of point * point * point
```

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

```
Circle ((3.0, 4.0), 2.0)
```

[42] ✓ 0.0s

```
- : figure = Circle ((3., 4.), 2.)
```

Sum type: option

```
type 'a option = None | Some of 'a
```

[23] ✓ 0.0s

```
type 'a option = None | Some of 'a
```

```
Some "Dwarf Knight"
```

[25] ✓ 0.0s

```
- : string option = Some "Dwarf Knight"
```

$$\frac{A}{A \vee B}$$

$$\frac{B}{A \vee B}$$

Pattern matching

- A value of a sum type can one of a set of alternatives.

$$\frac{A \Longrightarrow C \quad B \Longrightarrow C \quad A \vee B}{C}$$

- It is only possible to progress safely through case analysis.
- All branches must have the same type, akin to a conditional expression.

```
let string_of_point (x, y) = "(" ^ string_of_float x ^ ", " ^ string_of_float y ^ ")"

let string_of_figure f =
  match f with
  | Circle (p, r) → "Circle with center " ^ string_of_point p ^ " and radius " ^ string_of_float r
  | Rectangle (p1, p2) → "Rectangle with corners " ^ string_of_point p1 ^ " and " ^ string_of_point p2
  | Triangle (p1, p2, p3) → "Triangle with vertices " ^ string_of_point p1 ^ ", " ^ string_of_point p2 ^ ", and " ^ string_of_point p3
```

[5] ✓ 0.0s

```
val string_of_point : float * float → string = <fun>

val string_of_figure : figure → string = <fun>
```

Variants in C (Unions)

```
enum ShapeType {
    CIRCLE,
    RECTANGLE,
    TRIANGLE
};

union GeometricShape {
    enum ShapeType type;

    struct {
        double x;
        double y;
        double radius;
    } circle;

    struct {
        double x1;
        double y1;
        double x2;
        double y2;
    } rectangle;

    struct {
        double x1;
        double y1;
        double x2;
        double y2;
        double x3;
        double y3;
    } triangle;
};
```

```
union GeometricShape circle;
circle.type = CIRCLE;
circle.circle.x = 2.0;
circle.circle.y = 3.0;
circle.circle.radius = 5.0;

union GeometricShape rectangle;
rectangle.type = RECTANGLE;
rectangle.rectangle.x1 = 1.0;
rectangle.rectangle.y1 = 2.0;
rectangle.rectangle.x2 = 6.0;
rectangle.rectangle.y2 = 5.0;

union GeometricShape triangle;
triangle.type = TRIANGLE;
triangle.triangle.x1 = 1.0;
triangle.triangle.y1 = 1.0;
triangle.triangle.x2 = 4.0;
triangle.triangle.y2 = 5.0;
triangle.triangle.x3 = 7.0;
triangle.triangle.y3 = 2.0;
```

Case Classes and Pattern Matching in Scala

```
sealed trait class GenericShape

case class Circle(x: Double, y: Double, radius: Double) extends GenericShape
case class Rectangle(x1: Double, y1: Double, x2: Double, y2: Double) extends GenericShape
case class Triangle(x1: Double, y1: Double, x2: Double, y2: Double, x3: Double, y3: Double)
extends GenericShape

def printShapeDetails(shape: GeometricShape): Unit =
  shape match
    case Circle(x, y, r) => println(s"Circle: Center ($x, $y), Radius $r")
    case Rectangle(x1, y1, x2, y2) =>
       println(s"Rectangle: Top-left ($x1, $y1), Bottom-right ($x2, $y2)")
    case Triangle(x1, y1, x2, y2, x3, y3) =>
       println(s"Triangle: Vertex 1 ($x1, $y1), Vertex 2 ($x2, $y2), Vertex 3 ($x3, $y3)")
```

Pattern matching

```
let whatDoYouHave o =
  match o with
  | None -> "Nothing"
  | Some x -> "I have "^x
```

[ ]

```
let species_of p =
  match p.species with
  | Dog -> "Dog"
  | Cat -> "Cat"
  | Bird -> "Bird"
  | Fish -> "Fish"
```

[ ]

Pattern matching

```
type point = float * float

type figure =
  | Circle of point * float
  | Rectangle of point * point
  | Triangle of point * point * point
  | Polygon of point list
```

[41] ✓ 0.0s

```
let how_may_points_in f =
  match f with
  | Circle _ -> 0
  | Rectangle _ -> 4
  | Triangle _ -> 3
```

[46] ✓ 0.0s

```
File "[46]", lines 2-5, characters 2-19:
2 | ..match f with
3 |   | Circle _ -> 0
4 |   | Rectangle _ -> 4
5 |   | Triangle _ -> 3
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Polygon _

val how_may_points_in : figure -> int = <fun>
```

Pattern matching

```
type point = float * float

type figure =
  | Circle of point * float
  | Rectangle of point * point
  | Triangle of point * point * point
  | Polygon of point list
```

[41] ✓ 0.0s

```
let how_may_points_in f =
  match f with
  | Circle _ -> 0
  | Rectangle _ -> 4
  | Triangle _ -> 3
  | Polygon points -> List.length points
```

```
let do_you_like_this_figure f =
  match f with
  | Circle _ -> "Yes"
  | _ -> "No"
```

[ ]