Polymorphism in Functional Programming

In Unit2 we introduced the typed lambda-calculus and show its application in the type system of F#. We then proceeded to study built-in advanced types such as tuples and records. We concluded the unit by showing how to implement inheritance in functional programming through the use of immutable records and presented a complex case study. In this unit we show how to implement polymorphism in a functional programming language.

Discriminated Unions

In F# it is possible to specify a type as a discriminated union, which is essentially a type that can be constructed in multiple ways. In F# this can be achieved by using the following syntax

```
type T =
| Constructor_1 of T_1
| Constructor_2 of T_2
...
| Constructor_n of T_n
```

where `T` is the name of the type that we declare as usual, each `Constructor_i` is the name of each possible constructor that we can use to build our polymorphic type, and each `T_i` is the type of the input argument of the constructor. Note, if we need to pass multiple arguments to construct a specific case of the polymorhpic type we can provide a tuple. Note that the type argument is optional, so if we do not require arguments to construct that particular case, we can simply provide the constructor name only.

This would be equivalent to the following C# code:

```
interface T { }

class Constructor_1 : T
{
  public T_1 Data;
}

class Constructor_2 : T
{
  public T_2 Data;
}

...

class Constructor_n : T
{
```

```
  public T_n Data;
}
```

For instance, let us assume that we want to model a vehicle and each vehicle is characterized by a list of components. A car as four wheels, and an engine, a tank has two tracks, an engine,and a gun, a plane has an engine and two wings. This can be modelled by the following discriminated untion (assuming that we already have type definitions ofr `Wheel`, `Engine`, `Track`, and `Gun`):

```
type Vehicle =
| Car of Wheel * Wheel * Wheel * Wheel * Engine
| Tank of Track * Track * Engine * Gun
| Plane of Wing * Wing * Engine
```

Of course we can combine records and unions for better readability. For example we can refactor the code above as:

```
type Car =
  {
    Wheel1 : Wheel
    Wheel2 : Wheel
    Wheel3 : Wheel
    Wheel4 : Wheel
    Engine : Engine
  }

type Tank =
  {
    Track1 : Track
    Track2 : Track
    Engine : Engine
    Gun : Gun
  }
type Plane =
  {
    Wing1 : Wing
    Wing2 : Wing
    Engine : Engine
  }
type Vehicle =
| Car of Car
| Tank of Tank
| Plane of Plane
```

Note that we can add members to discriminated unions in the same way we do for records, for instance

```
type Vehicle =
| Car of Car
| Tank of Tank
| Plane of Plane
with
  member this.Run() = ...
```

Pattern Matching

Let us suppose that we want to give the implementation of the method `Run` described in the snippet above, so that each kind of `Vehicle` outputs a different sound. In order to do that, we must check which specific case of the vehicle polymorhpic type we are considering. In pure object-oriented programming this is usually achieved through a visitor or strategy design pattern. Several functional programming languages offer instead built-in abstractions to achieve the same behaviour. In F# this abstraction is called `match`:

```
match expression with
| Pattern_1 -> expr_1
| Pattern_2 -> expr_2
...
| Pattern_n -> expr_n
```

The `expression` argument can be any expression, including tuples, discriminated unions, or even function calls. The `match` control structure takes the expression argument and matches its pattern with one of those provided in its body. When the pattern is matched the corresponding right-hand side expression is computed. Assuming to have two patterns P_1 and P_2, we say that P_1 matches P_2 in the following cases:

- If P_2 is the wildcard symbol _ then P_1 always matches P_2.
- If P_1 is a variable then it always matches P_2. In this case we bind the value of P_1 to P_2.
- If P_1 is a value (like `5`,`true`,`"Hi!"`,...) then it matches P_2 only if $P_1 = P_2$. Note that this holds also for tuple and record values, which must be structurally equal.
- If P_1 is a discriminated union whose case is $C_1(x_1, x_2, ..., x_n)$ and P_2 is $C_2(y_1, y_2, ..., y_n)$ than P_1 matches P_2 if:
  - $C_1 = C_2$, i.e. P_1 and P_2 are the same case of the union (the name of the constructor in the discriminated union is the same).
  - x_i matches y_i for each $i = 1, 2, ..., n$, i.e. each argument of C_1 matches each argument of C_2.

The type checking of the `match` works as follows:

$\text{check}(\text{match } e \text{ with } |P_1 \rightarrow e_1|P_2 \rightarrow e_2|...|P_n \rightarrow e_n) \Rightarrow T \text{ when:}$

- $\text{check}(P_i) \Rightarrow T_1 \ \forall i = 1, 2, ..., n$
- $\text{check}(e) \Rightarrow T_1$
- $\text{check}(e_i) \Rightarrow T \ \forall i = 1, 2, ..., n.$

This means that the argument expression of `match` must have the same type of each pattern, and that the expression in the right-hand side of each case must have the same type. Note that if no case of the `match` passes the pattern matching, then you will have a runtime exception. The compiler of F# is able to detect automatically when this situation might occur, but it will only report it as a warning, since there might be cases when we are sure that the remaining combinations of patterns are never covered because of the execution flow of the program. So in principle not covering all the possible pattern cases is not a compilation error (so a wrong program), but it might happen that the pattern matching triggers a runtime exception because we forget to cover all the cases that we need.

Example:

Consider the two following type definitions:

```
type DegreeCourse =
  {
    Code              : string
    Name              : string
    EnrollmentDate    : int * int * int
  }

type Contract =
  {
    Serial              : string
    Salary              : float
    BeginDate           : int * int * int
  }

type Student =
  {
    Id          : string
    Name        : string
    LastName    : string
    Enrollment  : DegreeCourse
  }

type Employee =
  {
    Id          : string
    Name        : string
    LastName    : string
    Enrollment  : Contract
  }

type SchoolPerson =
```

```
| Student of Student
| Employee of Employee
```

and the following code:

```
let student1 =
  Student(
  {
    Id     = "0963963"
    Name   = "John"
    LastName = "Doe"
    Enrollment =
      {
        Code    = "INF"
        Name    = "Computer Science"
        EnrollmentDate = (15,08,2018)
      }
  })

match student1 with
| Employee e -> ...
| Student s -> ...
```

According to the rules above the first case of the match will fail to match the pattern, since the union case `Student` is different than the union case `Employee` in the pattern.

The second case will pass the pattern matching, since the union case is `Student` as well and the element of the pattern is a variable.

Now let us consider the following code:

```
match student1 with
| Employee e -> ()
| Student
    ( {
        Id = _
        Name = _
        LastName = _
        Enrollment = (
          {
            Code    = "INF"
            Name    = "Computer Science"
            EnrollmentDate = (15,08,2018)
          })
      }) -> ()
```

The second case of the match checks the structural equality of the record in `student1` with the one in the pattern: the fields `Id`, `Name`, and `LastName` are structurally equal since their value is the wildcard. The field `Enrollment` will be recursively checked for structural equality, since it contains a record instance. All the three fields of this record are structurally equal to those in `student1`, thus the record `Enrollment` is structurally equal both in the pattern of the `match` and in `student1`. Note that, to be absolutely accurate, we should also recursively check the structural equality of the tuples in `EnrollmentDate`, but for brevity we skip this step in this explanation. Given these considerations, we can conclude that `student1` passes the pattern matching. Informally, this case of the union is checking if `student1` contains either a school employee or a student enrolled in the Computer Science course in a specific date.

Options

One simple example of a discriminated union is the type `Option<T>`. An option represents an optional value, which is used in a situation when we are not sure whether we have a value or not. It is a safe version of `null`, because having a discriminated union will force you to explicitly handle the situation in which we do not have a value through pattern matching. The optional type is defined as :

```
type Option<'T> =
| None
| Some of T
```

Note that this type is already built-in in F# in the library `FSharpCore` that is automatically loaded by the compiler (so you do not have to import it explicitly).

Consider now a simulation of a server connection, where a server can fail to reply. A server is defined by an address, which consists of four bytes, a reply probability, which simulates the reliability of the server, and a reply message. A connection is built by providing the ip address of the server you want to connect to. The connection holds an optional `Data` field which contains a possible answer from the server. Note that this is optional because we cannot know if the server will ever reply or not. The connection has a method `Connect` that takes as input a `Server` and tries to connect to it and get a reply. The connection fails to get a reply if the address of the server does not match or if a random number is greater than the chance of getting a reply (we simulate a server failure). Note that to access the reply of the server we must pattern match the `Data` field against the possible pattern of `Option<T>`: if `Data` matches the pattern `Some data` then we access `data` contained in it (remember that, when an element of the pattern is a variable, we bind the value of the corresponding variable in the pattern that we are comparing from). You can see the implementation of this example below.

```
let r = System.Random()

type Server =
  {
    Address   : byte * byte * byte * byte
    ReplyProbability : float
```

```
    Reply     : string
  }
  with
    static member Create(address : byte * byte * byte * byte,replyProbability : flo
      {
        Address = address
        ReplyProbability = replyProbability
        Reply = reply
      }

type Connection =
  {
    Address   : byte * byte * byte * byte
    Data      : Option<string>
  }
  with
    static member Create(address : byte * byte * byte * byte) =
      {
        Address = address
        Data = None
      }
    member this.Connect (server : Server) =
      if this.Address <> server.Address ||
         r.NextDouble() > server.ReplyProbability then
         this
      else
        { this with Data = Some server.Reply }


let helloServer = Server.Create((169uy, 180uy, 0uy, 1uy),0.35,"Hello!")
let connection = Connection.Create((169uy, 180uy, 0uy, 1uy))
match connection.Connect(helloServer).Data with
| Some data -> printfn "The server replied: %s" data
| None -> printfn "404 server not found"
```

Lists

Lists are another data type that can be defined by using discriminated unions. We can think of a list as either an empty collection, or an element followed by a collection of elements (possibly empty). This element is usually defined as head. We can thus define a list as

```
type List<'a> =
| Empty
| List of 'a * List<'a>
```

Since the cases of a discriminated union are actually functions, it is possible to define custom operators to construct each case of the union. In F# the type for lists is defined as:

```
type List<'a> =
| ([])
| (::) of 'a * List<'a>
```

and it is automatically included, like `Option`. This is quite useful since we can use the operator `::` in a pattern matching case to extract the list head as `x :: xs`, and the symbol `[]` for the empty list. Lists can also be built by giving a list expression, like `[1;3;-8;7;2]`. Note that lists in functional programming are *immutable*, unlike their counterpart in OO programming. This means that it is not possible to set the value of their elements but only to read them. Another important operator in lists is the *concatenation* operator `@`. This operator takes two lists as input and merges them into one. For instance the expression :

```
[3;5;1;2] @ [1;3;4;1]
```

results into `[3;5;1;2;1;3;4;1]`.

Functions on lists are generally recursive, for example the following code computes the length of a list:

```
let rec length (l : List<'a>) : int =
  match l with
  | [] -> 0
  | _ :: xs -> 1 + (length xs)
```

Note that in the second case of the pattern we choose to use the wildcard symbol for the head of the list, since it is not required to compute the length. The tail of the list `xs`

Example

Let us compute the sum of the elements of a list. The base case of the recursion is when we have an empty list, whose elements add to 0. The recursive step will call `sum` on the tail of the list and add the result to the element in the head.

```
let rec sum (l : List<int>) : int =
  match l with
  | [] -> 0
  | x :: xs -> x + (sum xs)
```

Example

Let us implement the function `unzip` that, given a list `l` of pairs, creates a pair of lists `l1` and `l2`, where `l1` contains all the elements in the first item of the pairs in `l` and `l2` contains all the elements in the second item of the pairs in `l`.

Let us start by reasoning on the type signatures of the function: the input list is a list of pairs. There is no

restriction on the specific types of the elements of the pairs, so we use two generic types, which means `List<'a * 'b>`. The function returns a pair of lists, where the first one contains all the elements in the first item of each pair and the second all the elements in the second item. This means that the return type of the function is `List<'a> * List<'b>`.

The function is implemented recursively: the base case is of course an empty list, for which we return a pair of empty lists. The recursive case applies `unzip` to the tail of the list. This results in a pair of lists `l1` and `l2`. After this step has been computed we insert the first element of the pair in the head in front of `l1` and the second element in the front of `l2`.

```
let rec unzip (l : List<'a * 'b>) : List<'a> * List<'b> =
  match l with
  | [] -> ([],[])
  | (x,y) :: xs ->
      let l1,l2 = unzip xs
      x :: l1,y :: l2
```

Note the use of pattern matching to explicitly extract the pair in the head of the list, and also in the binding to explicitly bind to separate variables the two lists computed by the recursive step.

Example

Let us consider a list where each element can be either an atomic element or a nested list. We want to implement a function that flattens the elements of this list into a list with just one layer. For instance `[3;5;[4;3;[2;1;3];1];3;4;[1;2;3];6]` becomes `[3;5;4;3;2;1;3;1;3;4;1;2;3;6]`. Unfortunately in F# elements of a list are homogeneous, meaning that they must have the same type. Achieving our purpose requires to define a polymorphic type that can be either an atomic element or a nested list:

```
type ListElement<'a> =
| Element of 'a
| NestedList of List<ListElement<'a>>
```

Note that `NestedList` can recursively contain other nested lists, so the type definition is recursive.

Our `flatten` function will take as input a `List<ListElement<'a>>`, so that each element can be either a simple element or a nested list. It will return simply `List<'a>`, because, after it runs, all the elements will be at the same level of nesting. The base case of the recursion is an empty list, which will simply return an empty list. The recursive case must decide what to do depending on whether the head is a simple element or a nested list. In the case of a simple element we use pattern matching on both the list and the polymorhpic type `ListElement<'a>` to extract the content of `Element`. We recursively flatten the tail of the list and then we prepend the content of `Element` to it. The case of a nested list is slightly more complex: we apply pattern matching on the head of the list extracting the list itself from the polymorphic type. Whe then recursively flatten the nested list, we flatten the tail , and finally we concatenate the head flattened list with the flattened tail.

```
let rec flatten (l : List<ListElement<'a>> ) : List<'a> =
  match l with
  | [] -> []
  | Element x :: xs -> x :: (flatten xs)
  | NestedList nested :: xs ->
      let nestedFlatten = flatten nested
      nestedFlatten @ (flatten xs)
```

Tanks 2.0 with Polymorphism

In this section we refactor the implementation of *Tanks* given in the previous unit. We left pending the problem of letting fight each other tanks with 1 gun or with 2. At this point one solution would be to simply add a list of weapons to a tank instead of using two different types of tanks. This solution is, unfortunately, not optimal, because what we want to achieve is allowing only 1 or 2 weapons in a tank. Of course one could think of implementing a runtime check that forbids a list longer than 2, but this is again a bad solution because it would cause a runtime exception. By using polymorphism we can achieve the same result with a static check, which happens at compile time, thus completely preventing the generation of programs containing wrong tank configurations.

We start by refactoring the record `Gun` to directly contain a method to fire a weapon, which was initially defined by the `Tank` record:

```
type Gun =
  {
    Name : string
    Penetration : float
    Damage : float
  }
  with
    static member Create(name: string, penetration : float, damage : float) =
      { Name = name; Penetration = penetration; Damage = damage }
    member this.Shoot(tank : TankKind) =
      if this.Penetration > tank.Armor then
        let t = tank.Damage this.Damage
        printfn "%s shoots %s causing %f damage --> HEALTH: %f"
          this.Name
          tank.Name
          this.Damage
          t.Health
        t
      else
        let t = tank.Scratch this.Penetration
        printfn "%s shoots %s causing %f penetration --> ARMOUR: %f"
          this.Name
          tank.Name
          this.Penetration
```

```
            t.Armor
          t
```

The Tank and Tank2Weapons will now contain only a property to define when they are destroyed:

```
and Tank =
  {
    Name : string
    Weapon : Gun
    Armor : float
    Health : float
  }
  with
    static member Create(name :string, weapon : Gun, armor : float, health : float)
      {
        Name = name
        Weapon = weapon
        Armor = armor
        Health = health
      }
    member this.IsDead = this.Health <= 0.0

and Tank2Weapons =
  {
    SecondaryWeapon       : Gun
    Base                  : Tank
  }
  with
    static member Create (weapon : Gun,tank : Tank) =
      {
        SecondaryWeapon = weapon
        Base = tank
      }
    member this.IsDead = this.Base.Health <= 0.0
```

We now define a polymorphic type to express the fact that we can have either tanks with one weapon or two:

```
and TankKind =
| Tank of Tank
| Tank2Weapons of Tank2Weapons
...
```

This type will contain a series of utility properties to extract information about the tank independently of its kind, and a method that implements the logic of the fight, between two tanks of any kind.

```
and TankKind =
| Tank of Tank
| Tank2Weapons of Tank2Weapons
with
  member this.Name =
    match this with
    | Tank t -> t.Name
    | Tank2Weapons t -> t.Base.Name
  member this.Health =
      match this with
      | Tank t -> t.Health
      | Tank2Weapons t -> t.Base.Health
  member this.Armor =
    match this with
    | Tank t -> t.Armor
    | Tank2Weapons t -> t.Base.Armor
  member this.Damage(damage : float) : TankKind =
    match this with
    | Tank t ->
        Tank { t with Health = t.Health - damage }
    | Tank2Weapons t ->
        Tank2Weapons { t with Base = { t.Base with Health = t.Base.Health - damage
  member this.Scratch(damage : float) : TankKind =
    match this with
    | Tank t ->
        Tank { t with Armor = t.Armor - damage }
    | Tank2Weapons t ->
        Tank2Weapons { t with Base = { t.Base with Armor = t.Base.Armor - damage }
  member this.Fight(tank : TankKind) : TankKind * TankKind =
    let outcome (loser : TankKind) (winner : TankKind) =
      printfn "%s: KABOOOM!!! %s wins" loser.Name winner.Name
      if this = loser then
        this,tank
      else
        tank,this
    match this,tank with
    | Tank t1,Tank t2 ->
        if t1.IsDead then
          outcome this tank
        elif t2.IsDead then
          outcome tank this
        else
          let tank1 = t1.Weapon.Shoot(tank)
          let tank2 = t2.Weapon.Shoot(this)
          tank1.Fight tank2
    | Tank t1,Tank2Weapons t2
    | Tank2Weapons t2, Tank t1 ->
        if t1.IsDead then
          outcome this tank
```

```
    elif t2.IsDead then
      outcome tank this
    else
      let tank2 = t1.Weapon.Shoot(tank)
      let tank1 = t2.Base.Weapon.Shoot(this)
      let tank1 = t2.SecondaryWeapon.Shoot(tank1)
      tank1.Fight tank2
| Tank2Weapons t1, Tank2Weapons t2 ->
    if t1.IsDead then
      outcome this tank
    elif t2.IsDead then
      outcome tank this
    else
      let tank2 = t1.Base.Weapon.Shoot(tank)
      let tank2 = t1.SecondaryWeapon.Shoot(tank2)
      let tank1 = t2.Base.Weapon.Shoot(this)
      let tank1 = t2.SecondaryWeapon.Shoot(tank1)
      tank1.Fight tank2
```

The method will use pattern matching to determine what kind of tank combination we currently have. This is needed because a tank with one weapon will shoot only once, while a tank with two weapons will shoot twice, one for each weapon. So, for instance, in the case of a tank with one weapon fighting a tank with two weapons, we will let the first tank shoot once, and then the second twice. Also notice the use of pattern matching to merge the logic of `Tank` fighting `Tank2Weapons` and the symmetric case of a `Tank2Weapons` fighting a `Tank` into a single case. Be careful because, in this case, the following pattern would produce a compilation error

```
| Tank t1,Tank2Weapons t2
| Tank2Weapons t1, Tank t2 -> ...
```

because `t1` would be in one case be of type `Tank` and in the other of type `Tank2Weapons`. The same applies to `t2`.