Types in Functional Programming

In Unit 1 we introduced an untyped formulation of lambda calculus, which is the foundational model of functional programming. We then proceeded to translate the constructs of lambda calculus into the functional programming language F# without focusing on types. In this chapter we introduce a statically typed version of lambda calculus and then we show how its usage in F#. We then proceed to define basic data structures in F#

Typed Lambda Calculus

In this section we present the typing rules for lambda-calculus. As we have seen in the previous unit, lambda-calculus is made of the following main components: variables, abstractions, and function applications.

//TODO

Type Annotation in F#

Type checking in F# follows the same basics rules defined in the previous section. Type annotation of bindings can be achieved with the following syntax:

```
let (id : type) = expr
```

where `id` is the identifier used for the binding and `type` a valid type indentified. Basic types in F# are `int`, `float` (double-precision floating-point numbers like `double` in C#), `float32` (single-precision floating-point numbers like `float` in C#), `string`, ...

For example:

```
let (x : float) = 5.25
```

Note that F# does not perform automatic type conversions like C#, so for example the following code will produce a type error:

```
let (x : float) = 5
```

whereas, the same code in C# would work:

```
double x = 5;
```

In order to correctly type the expression above, we use the correct floating-point double-precision literal:

```
let (x : float) = 5.0
```

For this reason, every numerical type has its different corresponding literal in F#. A comprehensive list can be found in MSDN.

Arguments of lambda abstractions can be typed analogously:

```
let (f : t1 -> t2 -> ... -> tn -> tr) =
  (fun (arg1 : t1) (arg2 : t2) ... (argn : tn) -> expr) : t1 -> t2 -> ... -> tn
```

Notice that, if we do not want to rely on type inference, we must provide the type of the binding as the type of an abstraction, as we are binding a lambda. For example:

```
let (add: int -> int -> int) = fun (x : int) (y : int) -> x + y
```

As seen in Unit 1, an alternative notation to define functions is possible, which becomes with type annotations:

```
let f (arg1 : t1) (arg2 : t2) ... (argn : tn) : tr = expr
```

Thus, for example, the function add can be redefined as:

```
let add (x : int) (y : int) : int = x + y
```

F# supports generic polymorphism, thus we can provide type parameters to functions. This is done by adding an apostrophe before the type name, such as:

```
let (stringify : 'a -> string) = fun (x : 'a) -> string x
```

This means that the function `stringify` is generic with respect to the type parameter `'a`. Note that the language distinguishes between non-generic and generic types by checking if they are preceeded by an apostrophe, thus `'string` denotes a generic type called `string` and `string` (without the apostrophe) is the built-in type for strings.

Basic Data Structures in F#

F# natively implements complex data structures such as *tuples* and *lists*. A tuple is an ordered sequence of non-homogeneous values, such as `(3,"Hello world!",-45.3f)`. The type of a tuple is denoted as

```
t1 * t2 * ... * tn
```

where `t1`, `t2`,..., `tn` are types. Thus a tuple is the n-ary cartesian product of values of type `t1`, `t2`,..., `tn`. For example:

```
let (t : int * string * float32) = (3,"Hello world!",-45.3f)
```

Tuples, of course, can be passed as arguments to functions. In this context, there is a particular application of this which is an alternative to currying. In Unit 1 we saw that in lambda calculus (and also in F#) function admits one argument only. In order to model the behaviour of functions that operate on more than one argument, we relied on the notion of currying: a function that wants to use two arguments will simply return in its body another lambda that is able to process the second argument and has in its closure the first argument. For instance:

```
let add = fun x -> fun y -> x + y
```

When we call such function with `add 3 5` we replace `x` with 3 in its body thus generating `fun y -> 3 + y`, and then we apply `(fun y -> 3 + y) 5` thus obtaining `3 + 5 = 8`. An alternative to this is the *uncurried* version, where we pass the arguments in a tuple as follows:

```
let addUncurried = fun (x,y) -> x + y
```

Note that the curried and uncurried versions are not interchangable because their type is different. For instance, the type of `add` is:

```
let (add : int -> int -> int) = fun x y -> x + y
```

while the type of `addUncurried` is:

```
let (addUncurried : int * int -> int) = fun (x,y) -> x + y
```

Notice that the uncurried version of a function takes both arguments all together, thus partial application is not possible: we can call `add 3` and this will generate as result `lambda y -> 3 + y`, but we cannot call `addUncurried 3` because this would mean passing an argument of typ `int` to a function that expects `int * int`. We will see further ahead in this course that it is possible to define a generic function that can convert the curried version of a function to the uncurried version, and the opposite.

Records

Recors are a set of named values that can optionally define some members. This definition resembles that of Class in a object-oriented language, but there is a profound difference: the fields of a record are by default immutable, meaning that it is not possible to change their values directly. Of course, being a hybrid language, F# allows also to define mutable record fields, but, as said before, we ignore mutability in this course. A record is declared with the following syntax:

```
type R =
  {
    f1 : t1
    f2 : t2
    ...
    fn : tn
  }
```

For instance, the following record represents the login information to connect to a server:

```
type LoginInfo =
  {
    UserName        : string
    Password        : string
    Address         : string
  }
```

Synce F# is indentation sensitive, we must place particular care about how we indent the record definition: brackets should be indented with respect to the `type` keyword, and fields must be indented with respect to brackes. Failing to do so will result in a compilation error.

Optionally a record can define some methods and properties (members):

```
type R =
  {
    f1 : t1
    f2 : t2
    ...
    fn : tn
  }
  with
    static member M1(arg1 : t1, arg2 : t2, ..., argn : tn1) : tr1 =
    ...
    member this.M2(arg1 : t1, arg2 : t2, ..., argn : tn2) : tr2 =
    ...
```

A characteristic of F# is that an instance method must always declare the name of the implicit `this` parameter. This is so because F# does not restrict the name of the implicit parameter to a specific keyword, like `this` in C#, rather it can be customized, so for example you could call it `self` or `current`. Notice that it is possible also to use the curried version of methods with the usual syntax. For instance:

```
member this.M2 (arg1 : T1) (arg2 : T2) ... (argn : tn2) : tr2 = ...
```

Also notice that, in order to define a recursive method, it is not necessary to use the `rec` keyword as for functions. All members can be recursivelly called without the need of an extra clause.

A record can be instantiated with the following syntax:

```
let r = { f1 = value1 ; f2 = value2 ; ... ; fn = valuen }
```

or

```
let r =
  {
    f1 = value1
    f2 = value2
    ...
    fn = valuen
  }
```

It is immediately evident that, for records with a high number of fields, this syntax becomes quite cumbersome. For this reason, it is preferable to define a static method `Create` to instantiate a record:

```
type R =
  {
    f1 : t1
    f2 : t2
    ...
    fn : tn
  }
  with
    static member Create(arg1 : t1, arg2 : t2, ..., argn : tn) : R =
      {
        f1 = arg1
        f2 = arg2
        ...
        fn = argn
      }
```

Record copy and update

As said above, records are immutable, so it is not possible to directly update their fields. In order obtain the same effect of a field update, we must create a new record where all the values of the fields that are left untouched by the update are initialized by reading the corresponding values in the original record, and all the updated fields are initialized with the new value. For instance, let us consider the `LoginInfo` above, and suppose that you need to change the server address, that would require the following steps:

```
let oldLogin = { UserName = "awesomeuser@aw.us" ; Password = "supersecretkey"; A
let newLogin = { UserName = oldLogin.UserName; Password = oldLogin.Password; Adc
```

You can immediately notice that this operation becomes quite cumbersome when updating just a small number of fields of records with many fields. For this reason, the following shortcut is available:

```
let newRecord =
  { oldRecord with
      f1 = v1
      f2 = v2
      ...
      fk = vk
  }
```

This will make a copy of oldRecord and initialize the fields `f1`,`f2`, ..., `fk` with the specified values, while the others simply contain the values from `oldRecord`. The concrete example above becomes then :

```
let oldLogin = { UserName = "awesomeuser@aw.us" ; Password = "supersecretkey"; A
let newLogin =
  {oldLogin with
      Address = "165.40.69.69"
  }
```

Structural Equality

Lacking the notion of states, functional programming languages cannot perform equality comparisons based on references, like we are used to in imperative programming languages. By default, all data structures in F#, including newly-defined records, are compared by value and not by reference. This means that, using the = operator, F# will recursively check the components of the data structures and compare their value. For instance, consider the tuples `let t1 = (1,3,5)` and `let t2 = (1,3,-5)`: in this case F# will compare the first component of `t1` with the first component of `t2`, which passes the test. Then it will compare the second component of `t1` with the second of `t2`, passing the test as well. Finally, the third components are compare, which returns `false` because of course 5 and -5 are different. The same is done for the fields of a record. Notice that, since tuple components and record fields can contain complex data structures, this procedure is recursive, i.e. if the component/field is a tuple or a record then the structural equality is recursively applied on their values. For instance, the value in `test` in the following code is `true`:

```
let l1 = { UserName = "awesomeuser@aw.us" ; Password = "supersecretkey"; Address
let l2 = { UserName = "awesomeuser@aw.us" ; Password = "supersecretkey"; Address
let test = l1 = l2
```

Case Study: Tanks and Guns

In this section we present a small case study to show the usage of records. Let us assume that we want to model an entity `Tank` defined by name, speed, weapon, armor, and health. Each tank weapon is a gun defined by name, penetration power, and damage. A tank can shoot another tank with its gun, with the following effect: if the gun penetration is higher than the armour than the health of the target is reduced by the weapon damage. Otherwise the amount of armour is decreased by the gun penetration. Let us first define the records for guns and tanks:

```
type Gun =
  {
    Name              : string
    Penetration       : float
    Damage            : float
  }
  with
    static member Create(name: string, penetration : float, damage : float) =
      { Name = name; Penetration = penetration; Damage = damage }

type Tank =
  {
    Name    : string
    Weapon  : Gun
    Armor   : float
    Health  : float
  }
  with
    static member Create(name :string, weapon : Gun, armor : float, health : flc
      {
        Name = name
        Weapon = weapon
        Armor = armor
        Health = health
      }
```

and let us define some gun and tank models:

```
let kwk36 = Gun.Create("88mm KwK 36", 150.0, 90.0)
let f32 = Gun.Create("76mm F-32", 70.0, 60.0)
let kwk40short = Gun.Create("75mm kwk 37", 35.5, 55.5)
let kwk40Long = Gun.Create("75mm KwK 40", 99.5, 55.5)
let m1a1 = Gun.Create("76mm M1A1", 99.0, 60.0)
let tiger = Tank.Create("Pz.Kpfw. VI Tiger Ausf. E", kwk36, 340.0, 800.0)
let t34 = Tank.Create("T-34/76", f32, 200.0, 400.0)
let p4f = Tank.Create("Pz.Kpfw. IV", kwk40short, 130.0, 350.0)
let p4g = Tank.Create("Pz.Kpfw. IV", kwk40Long, 130.0, 350.0)
let shermanE8 = Tank.Create("M4A3 Sherman E8", m1a1, 220.0, 450.0)
```

Now let us implement the logic of the fight as a method of `Tank`. This method will take as explicit parameter the opponent tank.

```
member this.Shoot (tank : Tank) = ...
```

This method will have to check the weapon penetration of `this` against the `Armor` of `tank`. If it is higher than we print a message on the status and we update the health of `tank`. If it is lower than we reduce the armour value of `tank`.

```
member this.Shoot(tank : Tank) =
  if this.Weapon.Penetration > tank.Armor then
    printfn "%s shoots %s with %s causing %f damage --> HEALTH: %f"
      this.Name
      tank.Name
      this.Weapon.Name
      this.Weapon.Damage
      tank.Health
    { tank with Health = tank.Health - this.Weapon.Damage }
  else
    printfn "%s shoots %s with %s reducing armour by %f --> ARMOUR: %f"
      this.Name
      tank.Name
      this.Weapon.Name
      this.Weapon.Penetration
      tank.Armor
    { tank with Armor = tank.Armor - this.Weapon.Penetration }
```

Now let us make two tanks fight. We do so by implementing a function that takes two tanks and repetedly calls the `Shoot` method in turn until one of the two tanks is destroyed. This function will be recursive, since it must repeat the shooting phase an indefinite number of times. The base case of the recursion is when one of the two tanks is destroyed. Ohterwise we must call the function again with the updated tanks after they shoot each other:

```
let rec fight (t1 : Tank) (t2 : Tank) =
  if t1.Health <= 0.0 then
    printfn "%s: KABOOOM!!! %s wins" t1.Name t2.Name
    t1,t2
  elif t2.Health <= 0
    printfn "%s: KABOOOM!!! %s wins" t2.Name t1.Name
    t1,t2
  else
    let t2 = t1.Shoot t2
    let t1 = t2.Shoot t1
    fight t1 t2
```

Now let us assume that we want to retrofit a specific tank with a better gun. The retrofit takes a gun and tries to mount it on a tank. If the tank model matches the retrofitting model then the new gun is applied, otherwise the tank is returned as it is. In order to do this, we have to exploit the structural equality provided by F#: we compare the current tank with the retrofitting model, and if they are structurally equal then we returned a new tank with the modified gun, otherwise we return the tank as it is.

```
member this.Retrofit (gun : Gun, tank : Tank) =
  if this = tank then
    {this with
        Weapon = gun}
  else
```

```
    this
```

Structural equality also offers the possibility of refactoring our code. Notice that in `fight` the first two cases of the `if` are doing the same thing. We can thus define a function **nested** into `fight` that prints the message in both cases and returns `t1` and `t2`. This function will be defined as

```
let outcome (loser : Tank) (winner : Tank) = ...
```

This function will be called as `outcome t1 t2` in the first case of the `if` and as `outcome t2 t1` in the second case. Notice that, since the argument in the two calls are swapped, in the body of `outcome` we cannot simply return `loser,winner` because that would sometimes swap the returned tanks. We can check if `loser` is indeed `t1` and, if not, return `t2,t1` instead of `t1,t2` (by convention we are returning the `loser` tank in the first position of the tuple).

```
let outcome loser winner =
  printfn "%s: KABOOOM!!! %s wins" loser.Name winner.Name
  if t1 = loser then
    t1,t2
  else
    t2,t1
```

With this refactoring, `fight` becomes:

```
let rec fight (t1 : Tank) (t2 : Tank) =
  let outcome (loser : Tank) (winner : Tank) =
    printfn "%s: KABOOOM!!! %s wins" loser.Name winner.Name
    if t1 = loser then
      t1,t2
    else
      t2,t1
  if t1.Health <= 0.0 then
    outcome t1 t2
  elif t2.Health <= 0.0 then
    outcome t2 t1
  else
    let t2 = t1.Shoot t2
    let t1 = t2.Shoot t1
    fight t1 t2
```

Inheritance via record nesting

Inheritance is an important feature of object-oriented programming that allows, among other things, to recycle the code and the definition of existing classes and, at the same time, to enrich them with additional functionalities. F# records cannot be inherited, but it is possible to achieve an analogous result by nesting records. Consider again the record `Tank` used above, and suppose that we want to define a new kind of tank with more than one weapon. This would be expressed in C# as

`Tank2Weapons : Tank`. In F# we can define a new record `Tank2Weapons` that has a field of type `Tank` containing the base record `Tank`. This new tank will have an additional field defining the secondary gun and a new way of shooting: it will first shoot the main gun of the tank and then shoot the secondary gun.

```
type Tank2Weapons =
  {
    SecondaryWeapon   : Gun
    Base              : Tank
  }
  with
    static member Create (weapon : Gun) (tank : Tank) =
      {
        SecondaryWeapon = weapon
        Base = tank
      }
```

Now let us refactor the `Shoot` function and let us define it in `Gun` instead of `Tank`, so that it becomes:

```
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 : Tank) =
      if this.Penetration > tank.Armor then
        printfn "%s shoots %s with %s causing %f damage --> HEALTH: %f"
          this.Name
          tank.Name
          this.Name
          this.Damage
          tank.Health
        { tank with Health = tank.Health - this.Damage }
      else
        printfn "%s shoots %s with %s reducing armour by %f --> ARMOUR: %f"
          this.Name
          tank.Name
          this.Name
          this.Penetration
          tank.Armor
        { tank with Armor = tank.Armor - this.Penetration }
```

and let us also redefine `fight` as member of both `Tank` and `Tank2Weapons`:

```
//Fight in Tank
```

```
member this.Fight(tank : Tank) =
  let outcome loser winner =
    printfn "%s: KABOOOM!!! %s wins" loser.Name winner.Name
    if this = loser then
      this,tank
    else
      tank,this
  if this.Health <= 0.0 then
    outcome this tank
  elif tank.Health <= 0.0 then
    outcome tank this
  else
    let t2 = this.Weapon.Shoot tank
    let t1 = tank.Weapon.Shoot this
    t1.Fight t2

//Fight in Tank2Weapons
member this.Fight(tank : Tank2Weapons) =
  let outcome loser winner =
    printfn "%s: KABOOOM!!! %s wins" loser.Base.Name winner.Base.Name
    if this = loser then
      this,tank
    else
      tank,this
  if this.Base.Health <= 0.0 then
    outcome this tank
  elif tank.Base.Health <= 0.0 then
    outcome tank this
  else
    let t2 = { tank with Base = this.Base.Weapon.Shoot tank.Base }
    let t2 = { t2 with Base = this.SecondaryWeapon.Shoot t2.Base }
    let t1 = { this with Base = tank.Base.Weapon.Shoot this.Base }
    let t1 = { t1 with Base = tank.SecondaryWeapon.Shoot tank.Base }
    t1.Fight t2
```

The version of `Fight` in `Tank2Weapons` will first shoot the base weapon of the current tank. This will return an updated copy of `Base` of the other tank, which must replace the current value in the `Base` field. Then it will shoot the secondary weapon thus obtaining another copy of `Base` that must replace the old value again. The same operations are performed for the second tank.

The attentive reader will notice that now we have a design problem: we can let `Tank` fight another `Tank` and `Tank2Weapons` fight `Tank2Weapons` but we cannot mix them up (as it would make sense). This problem can be solved by using polymorphism, thus by defining a function that accepts a `TankKind` that can be either a `Tank` or a `Tank2Weapons`, or function records, but we will explain these topics further ahead.