Introduction to functional programming

Functional programming is a programming paradigm that is profoundly different from the imperative paradigm. This programming paradigm grants additional properties on the code that help the programmer to write code that is maintainable and correct. For this reason also imperative languages widely used in the industry have been including functional programming construct during the last decades: C#, C++, Java, Javascript, Typescript all provide means of writing programs in terms of functional programming abstractions, and some of them even integrated type systems that are as expressive as those of Haskell or CamL. It is evident that functional programming trascendended the role of a tool used only by researchers and pioneers of programming and is becoming more and more a reality in the industry.

The goal of this unit is to show the reader a comparison between the model of imperative programming, which was explained during the Development courses of the first year, and the functional paradigm. We then proceed to define the semantics of a functional programming language, namely F#, and we present its main features.

Imperative programming vs functional programming

An imperative program is made of a sequence of instructions or commands that alter the memory, which is in general refered to also as *State*. In the introductory programming courses of the first year we said that a program is evaluated through its *semantics*, that is a set of rules defining the behaviour of each construct in the language. In general, in an imperative programming language, each of these rules processes a program and in general returns a new program and a new state.

$$eval(\langle P \rangle, S) \Rightarrow \langle P' \rangle, S'$$

The new program is generally returned because the evaluation of a single instruction might, in general, require multiple steps. The new state is due to the fact that some of the instructions might perform operations in memory and thus change the state.

In the following more advanced courses we further extended the model of state by allowing *scoping* and differentiate between *stack* and *heap*, but the underlying logic does not change in its substance. Indeed the evaluation of a program with stack and heap is given by the following rule:

$$eval(\langle I; J \rangle, S, H) \Rightarrow \langle I'; J \rangle, S', H' \text{ where } eval(\langle I \rangle, S, H) \Rightarrow \langle I' \rangle, S', H'$$

In this context the program uses a shared memory that is read and written by each instruction. This means that the order that we use to execute the instructions change the behaviour of the program. So two different programs with the same instructions in different order might produce two different results. For example consider the following two code snippets in Python:

```
x = 5
```

```
y = 2
x = x + 1
y = x - y
```

```
x = 5
y = 2
y = x - y
x = x + 1
```

In the first case, at the end of the execution, the state is $\{x := 6, y := 4\}$ while in the second case the state is $\{x := 6, y := 3\}$. The different results are determined by the different order of execution of line 3 and 4 in the two programs, wich is inverted in the second version. This creates undesirable side effect not only in single-thread programs but also in multi-thread ones. Getting rid of the problems related to shared memory often grants additional correctness properties and benefits, for example, parallelization and code maintainability. One very straightforward solution is getting rid of the mutable state itself, and this is where functional programming comes into play.

The computation in functional programming does not rely on the notion of state, and for this reason it is defined as *stateless* or *immutable*. The computation in a functional programming language consists of expression evaluations. Recalling the development courses of last year, we defined in the semantics a special semantics, called `eval_expr`, to process expression: its evaluation did not change the state of the program (even though we were in the context of imperative programming) because the result was simply a value resulting from a computation.

$$eval\_expr(\langle E \rangle, S) \Rightarrow \langle E' \rangle$$

The stateless model of computation grants an important property called *referential transparency*. Referential transparency grants that, at any point of a program, we can replace an expression with its value without changing the final result of the computation. In the following program

```
x = 5
x = x - 2
y = x + 3
```

we cannot replace, for instance, x with 5, which is the value we assign to it in the first line. For example, if we replace x with 5 in the last line of code, y would get the value 8. But this is clearly wrong, because if we execute all the statements in the program, at line 2 the value of x becomse 3, and thus y gets 6 after executing line 3. In a referential transparent program this does not happen: we are sure that if, at any moment, we know already the value of an expression and we replace it with its value we do not alter the result of the program execution. A consequence of this is also that, for instance, calling two different functions in a different order does not change the final result of a program, while in general this is not true in imperative programming.

Being intrinsecally different from imperative programming, functional programming will require a completely different computational model which is not based on instructions and state transitions. This underlying computational model is called *Lambda Calculus*.

Untyped lambda calculus

Lambda calculus is a computational model that is at the foundation of the semantics of all functional programming languages. Lambda calculus is radically different from the model above, but it has the same expressive power. This means that everything that we can compute in an imperative language can be computed also in a functional language. In this section we focus on its untyped version but later on we will extend it with a type system. Since the focus of this course is functional programming and not a theoretical course on lambda calculus we will skip some details. For example we will assume that the encoding of numbers and boolean values is already available in lambda calculus, and that arithmetic and boolean operations are possible, without formally showing their implementation in lambda calculus itself. We will just compute the result of arithmetic and boolean experssion as we would normally do.

Lambda calculus is made of three syntactic elements:

- *Variables* as x, y, A, ...
- *Abstractions*, that are function declarations with **one** parameter, in the form $\text{fun } x \to t$ where x is a variable and t is the body of the function.
- *Applications* (calling a function with one argument) in the form $t\ u$ where t is a function being called and u is its argument. Both t and u can be themselves complex expressions in lambda calculus.

Note that in lambda calculus abstractions can be passed as values, thus functions can be passed as arguments of other functions.

Now that we have defined the syntax of the language, we have to define its *semantics*, that is how its elements behave. This will be done through evaluation rules, in the same fashion as we did in previous development courses.

Variables

$$eval\ x \Rightarrow x$$

This rule states that variables cannot be further evaluated.

Abstractions

$$eval\ (\text{fun}\ \ x \to t) \Rightarrow (\text{fun}\ x \to t)$$

This rule states that abstractions **alone** cannot be further evaluated.

Substitution rule

$$eval\ (\text{fun}\ x \to t)\ u \Rightarrow t[x \mapsto u]$$

This rule states that whenever we call an abstraction with a parameter u, the result is the body of the abstraction where its argument variable is replaced by u

Example

$$eval\ (\text{fun}\ x \to x)\ A \Rightarrow x[x \mapsto A] \Rightarrow A$$

$$eval\ (\text{fun}\ x \to \text{fun}\ y \to x)\ (\text{fun}\ x \to x) \Rightarrow$$

$$(\text{fun}\ y \to x)[x \mapsto (\text{fun}\ x \to x)] \Rightarrow$$

$$\text{fun}\ y \to (\text{fun}\ x \to x)$$

Function application

$$eval\ (t\ u) \Rightarrow v \text{ where } eval\ t \Rightarrow t', eval\ u \Rightarrow u', eval\ t'\ u' \Rightarrow v$$

This rule states that, whenever we have a function application where the left term is not immediately an abstraction, we have to evaluate the terms left-to-right and then use their evaluation to compute the final result.

Example

$$eval\ ((\text{fun}\ x \to x)\ A)\ ((\text{fun}\ y \to A)\ B)$$

$$eval\ A\ ((\text{fun}\ y \to A)\ B)$$

$$A\ A$$

Currying

As defined above, abstractions can accept only one input parameter. This might look limiting, since in all programming languages it is possible to write functions that accept multiple parameters. However, in functional programming, we can return other abstractions as result of an abstraction to overcome this limitation. Consider the following method in C#:

```
static int Add(int x, int y)
{
  return x + y;
}
```

Assuming that arithmetic operators are available in lambda calculus (see the remark below for this), we can build a function that takes x as parameter and whose body contains **another abstraction** that accepts y as parameter and finally computes `x + y` in its body:

$$\text{fun}\ x \to \text{fun}\ y \to x + y$$

Now, let us actually test that this behaves as expected:

$$((\text{fun } x \to \text{fun } y \to x + y)\ 5)\ 3$$

$$(\text{fun } y \to 5 + y)\ 3$$

$$5 + 3$$

This mechanism is known as *currying*: the arguments to the abstraction are substituted one by one and replaced sequentially. Since the outer abstraction body contains another lambda, when we perform the first substitution, the result will be another abstraction whose body contains the inlined value 5 and the argument variable y. At the next function application, the parameter y of this second abstraction will be replaced by 3.

This can be seen as a mechanism where the outer abstraction can produce infinitely many abstractions as result that are capable to add any number to 5 (but to 5 only!).

For compactness, from now on, the following verbose notation

$$\text{fun } x_1 \to \text{fun } x_2 \to \ldots \to \text{fun } x_n \to t$$

will be shortened into

$$\text{fun } x_1\ x_2\ \ldots\ x_n \to t$$

so we will not write the internal arrows and fun symbols.

Bindings

Let us consider the following lambda calculus expression:

$$((\text{fun } f\ g \to f(\text{fun } y \to A)\ (\text{fun } g \to g) \to A)\ ((\text{fun } y \to A)\ (\text{fun } g \to g)) \to A)$$

The expression $(\text{fun } y \to A)\ (\text{fun } g \to g) \to A$ appears twice. Since it is inconvenient to write such a complex expression every time we need it, we would like to be able to write someting like:

$$\text{let E} = (\text{fun } y \to A)\ (\text{fun } g \to g) \to A \text{ in } ((\text{fun } f\ g \to f\ E)\ E)$$

This construct is called *binding*. Note that this does *not* (notice the emphasis on this word) have the same semantics as variable assignment in imperative programming: a variable assignment changes the state of a program, whilst here there is no change of state (we do not have a state at all) and it is merely a renaming of expressions.

We now show that bindings can be expressed in the current semantics of lambda calculus without adding something new. When we have a binding in the form

```
let x = y in t
```

we replace every occurrence of x in t with y. This allows us to use a shorter "version" x in t that will be replaced by its full form y when we need to evaluate the lambda calculus program. This is not much different than how we normally substitute variables in the substitution rule of lambda calculus. Actually, it is just the same! Thus, the semantics of `let` can be just expressed in terms of the substitution rule:

$$eval \text{ let } x = y \text{ in } t \Rightarrow eval \ (\text{fun } x \rightarrow t) \ y$$

Shadowing

Consider the following lambda calculus expression:

$$(\text{fun } x \ y \ x \rightarrow y \ x) \ A$$

One could blindly apply the substitution rule and end up with

$$\text{fun } y \ x \rightarrow y \ A$$

This is, however, incorrect: the scope of the outer variable x and the inner variable x is different. The x that we incorrectly substituted above is referring to the inner x parameter and not the outer. In other words, the inner x *shadows* the outer x. The correct result will, thus be

$$\text{fun } y \ x \rightarrow y \ x$$

For better clarity, one should imagine that the expression presented at the beginning of this paragraph was written as:

$$\text{fun } x_1 \ y \ x_2 \rightarrow y \ x_2$$

where it is shown explicitly that the outer parameter is different than the inner.

A final note on lambda calculus

Bindings are one of the many examples of language extensions that can be implemented directly in lambda calculus as it is. They are just syntax to make the language more usable but lambda calculus in its current form has the same expressive power as any regular programming language. Since this is not a theoretical course on formal computational models, we will not show here how to map every construct that we explain to lambda calculus, but we will merely acknowledge that this is possible. For instance, it is possible to build numbers, boolean, arithmetic operations, boolean operators, and conditional expressions (if-then-else) only in terms of abstractions, but we will just accept that this is possible without proving it and use them in functional languages and use them right away with their usual semantics. The only thing that we will add is a type system to lambda calculus.

F# vs Lambda Calculus

In this section we proceed to introduce F#, the language of choice for this course, by showing how the language abstractions of lambda calculus are mapped to it. F# is a hybrid functional-imperative language. This means that in F# we can combine abstractions from functional programming with imperative

statement. In this course we use its imperative part only to display something in the standard output. We choose this language over pure functional languages like Haskell because I/O operations (and not only) in such languages require the knowledge of advanced functional design patterns called *Monads* that go beyond the scope of this course.

Moreover, since we have not introduced typed lambda calculus yet, we will rely on the F# ability to *infer* types in the program that we write: in many functional languages type annotation is not mandatory (like, for instance, in Java or C) because the type checker of the compiler is powerful enough to understand the type of an expression most of the times (but not always, this is why it is important to manually determine the type of an expression).

Bindings

Bindings follow the same syntax we have defined in lambda calculus, for example:

```
let x = 5 in x + 1
```

It is possible to omit the `in` keyword by breaking the line, so that the binding above becomes:

```
let x = 5
x + 1
```

Note that F# is indentation-sensitive, which means that indentation is not just a way of making your code more readable but a syntax rule itself. In the case of a binding, indentation is necessary when the expression that is bound appears in a new line. For instance, if we wanted to put 5 on a new line we would need to indent the code:

```
let x =
  5
x + 1
```

In this case this looks quite useless, but when we have longer expressions this could be quite convenient:

```
let x =
  3 + 5 * 2 - % 4
x + 1
```

Abstractions and Function Applications

Abstractions can be defined by using the keyword `fun` as well. For example, the lambda calculus abstraction $\text{fun } x \to x + 1$ would become in F#:

```
(fun x -> x + 1) 3
```

Naturally, if we bind an abstraction we can then apply it by using the name of the binding, such as:

```
let inc = fun x -> x + 1
in
  inc 3
```

or in the more compact notation:

```
let inc = fun x -> x + 1
inc 3
```

which in lambda-calculus would become $(\text{fun } inc \rightarrow inc\ 3)\ (\text{fun } x \rightarrow x + 1)$. Note that shadowing works as expected in F#, so the result of the following program

```
let f = fun x -> x + 1
let f = fun x -> x - 1
f 3
```

will be 2 and not 4. An alternative syntax to defining abstractions is the following:

```
let inc x = x + 1
inc 3
```

Conditionals

Conditionals in functional programming have a different semantics then in imperative programming. In the previous courses of development we saw that imperative languages support two different kinds of conditionals: one is a conditional *statement* and one is a conditional *expression*. Since in functional programming we have no state, the only supported conditional is the conditional expression. F# support a conditional expression with the syntax:

```
if condition then
  thenExpr
else
  elseExpr
```

which is much different from, for instance, its C# counterpart

```
condition ? thenExpr : elseExpr
```

but they share the same semantics:

`eval <if C then T else E> ⟶ <if C' then T else E> where eval <C> ⟶ C'`

`eval <if true then T else E> ⟶ <T>`

`eval <if false then T else E> ⟶ <E>`

The following function shows the usage of the conditional expression:

```
let toInfinityAndBeyond =
  fun x ->
    if x >= 0 then
      x + 1
    else
      x - 1
```

Remember that conditional expressions always need and `else` expression.

Recursion

In functional programming there is no concept of loop. Indeed loops require stateful computation, as their semantics is defined as:

`eval (<while C do T>,S) ⟶ (<if C then { T; while C do T } else { done }>, S)`

Functional programming can achieve an equivalent behaviour through recursive functions. In F# recursive functions are defined through the keyword `let rec`. For example, let us assume that we want to return a string containing all the numbers divisible by a number n. We can achieve this through a recursive function: this function starts from 1 and tests all the numbers less or equal than n one by one. The base case of the recursion happens when the number that we are testing is greater than the current number. In this case we simply return an empty string. Otherwise we check if the number is divisible by n and, if that is the case, we add it to the string:

```
let rec getDivisors =
  fun n i ->
    if i > n then
      ""
    else
      if (n % i = 0) then
        i + " " + (getDivisors n (i + 1))
      else
        getDivisors n (i + 1)
```

Note that this function is equivalent to the following imperative code in C#:

```
static string GetDivisors(int n)
{
  int i = 0;
  string s = "";
  while (i <= n)
  {
    if (n % i == 0)
      s += " " + i;
  }
  return s;
}
```

Observe how, in the case of the recursive function, we use the stack of the recursive calls to carry ahead all the information necessary for the computation instead of relying on side effects and the state. This should give the reader an idea on why recursion and loops are two sides of the same coin, and why functional programming is as computationally expressive as imperative programming. The recursive function can then be invoked, for example, as:

```
getDivisors 15 1
```

This approach is not ideal, since the second parameter that we provide in the function application should always be 1, but we ask the caller to provide it. The recursive function above can be encapsulated within another function: this function takes as parameter only the number n and defines **internally** the code of the recursive function, that can just be invoked every time by passing n and 1.

```
let getDivisors =
  fun n ->
    let rec mkDivisors =
      fun n i ->
        if i > n then
          ""
        else
          if (n % i = 0) then
            i + " " + (getDivisors n (i + 1))
          else
            getDivisors n (i + 1)
    mkDivisors n 1
```

Printing to the standard output

As said above, F# is a hybrid objected-oriented/functional programming language, thus it supports also side effects. In this course we rely on side effects only to print something to the standard output. In F# it is possible to print to the standard output in two different ways: the first is to rely, as for C#, on `System.Console.WriteLine`, while the other is by using the functions `printf` or `printfn`, which is more idiomatic.

The function `printf` (and in the same `printfn`, with the only difference that a new line is added at the end) takes as input a *formatted string*. A formatted string contains regular text, as well as formatting information, that allows to inline the string representation of its parameters. The function can take a variable number of parameters, so it is possible to pass as many parameters as necessary. For example:

```
printf "The city of %s was founded in %d %s and is %f KM far from Rotterdam"
  "Rome" 748 "BC" 1615.0
```

One useful format parameter is `%A`, which is basically capable of generating a string for each data structure defined in F# (unless it is a `class`, but we are not going to use them in this course).

For a more comprehensive list of formatting parameters, visit MSDN

Summary

In this unit we started by describing the differences between imperative and functional programming. We showed that functional programming involves *stateless computation* as functional program consists of a sequence of expressions that are evaluated, rather than a sequence of instructions that change a state. We hinted that this has the benefit that the result of a program never depends on the order of evaluation of its expressions or function calls.

We then proceeded to outline the model that embraces all functional programming languages, called *lambda calculus*. In this unit we presented the semantics of the untyped version of this model.

We then introduced the functional programming language F# and show how to map constructs from lambda calculus in it.

Exercises

Exercise 1

Implement a function

```
let allNumber (n: int) : string =
```

that returns a string containing all numbers from 0 to n. Separate the numbers with a white space.

Exercise 2

Implement a function

```
let allNumberRev (n: int) : string
```

that returns a string containing all numbers from n to 0. Separate the numbers with a white space.

Exercise 3

Implement a function

```
let allNumberRange (lower: int) (upper: int) : string
```

that returns a string containing all numbers from n to 0. Separate the numbers with a white space.

Exercise 4

Implement a function

```
let allNumberRangeRev (lower: int) (upper: int) : string
```

that returns a string containing all numbers between lower and upper in reverse order. Separate the numbers with a white space.

Exercise 5

Implement a function

```
let allEvenRange (lower: int) (upper: int) : string
```

that returns a string containing all even numbers between lower and upper. Separate the numbers with a white space.

Exercise 6

Implement a function

```
let drawLine (length: int) : string
```

that returns a string containing length asterisks.

Exercise 7

Implement a function

```
let drawSymbols (symbol: char) (length: int) : string
```

that returns a string containing length repetitions of symbol.

Exercise 8

Implement a function

```
let toBinary (n: int) : string
```

that returns a string containing the binary representation of the input number (it must be positive). The binary representation is obtained using the following procedure:

1. Add to the end of the string the remainder of the division between the current number and 2.
2. Repeat the previous step with `n / 2` until the number is 0. In this case simply don't add anything.

Exercise 9

Implement a function

```
let toBase (n: int) (base: int) : string
```

that returns a string containing the representation of the input number in an arbitrary base (the number must be positive). The algorithm is the same as above except you must take the remainder of `n` divided by `base` and pass `n / base` to the next step.