Core aspects of PL design

If, on the one hand, the advances in technology have had the impact we just discussed in powering PL evolution, on the other hand the design of programming languages supporting the desired abstractions has always been, and still is, anchored to few core aspects and key principles:

- the **computation model** (or paradigm) adopted, whether imperative, functional, logical, object-oriented, concurrent;
- the **data abstraction mechanisms** available for data representation;
- the **typing discipline** provided to support program development
- the **memory management** and other low level tools employed to support an effective use of the computing resources available for the computation.

Each of these core aspects provides an independent dimension for programming language categorization. Among them, the first and most fundamental design decision concerns the computation model, and specifically the choice of whether it should be imperative or not.

Imperative vs functional programming

Imperative languages are basically high-level, complex versions of the von Neumann computer.

In its simplest form a von Neumann computer has three parts:

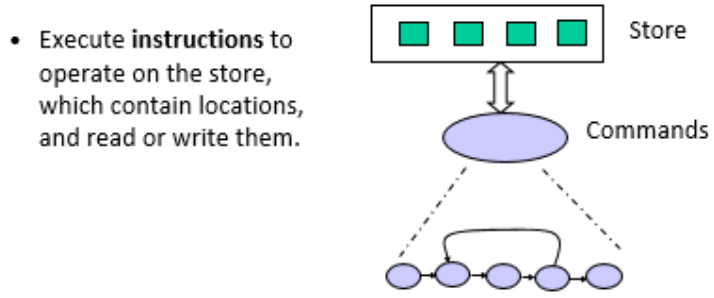
- a central processing unit (or CPU),
- a store,
- a bus connecting the CPU and the store.

Correspondingly, a program in an imperative language has an implicit state (i.e., values of variables, program counters, etc.) that is modified (i.e., side-effected) by constructs (i.e., commands, or instructions) in the source language. Such languages generally have an explicit notion of sequencing (of the commands) to permit precise and deterministic control of the state changes.

These are the "conventional" or "von Neumann languages" discussed by Backus in his 1977 Turing Award Address. They are well suited to traditional computer architectures, and to express *how*

something is to be computed (as opposed to *what* is being computed).

The following picture captures the essence of imperative computations.



The assignment statement is the core instruction, whose effect is to alter the underlying implicit store so as to yield a different binding for a particular variable. The *begin* . . . *end* construct is the prototypical sequencer of commands, as are the well-known conditional statement (qualified sequencer), and while loop (an example of a structured command). With these simple forms, we can write programs like the one below:

```
n := x;
a := 1;
while n > 0 do
 begin
 a := a*n;
 n := n-1
 end;
```

A quick look at the above code fragment is enough to tell that it computes the factorial of `x`. However, providing a compelling (i.e. mathematically grounded) argument that the program does indeed compute *what* we expect it to compute requires some effort. One way to accomplish that is by observing that the loop satisfies the following *invariant* condition: `{ a * n! == x! }`. To prove that, we can show that the condition is verified at each iteration of the loop:

- at the first iteration, the condition is trivially true as `a == 1` and `n == x`,
- at each further iteration, if we assume that `{a * n! == x!}` before executing the body of the loop, than it is immediate to see that the condition also holds at the end, as `a*n! == (a*n)*(n-1)!`

At the end of the last iteration the invariant is still valid, and `n==0`: then since `0! == 1`, from the invariant we have `a == x!` as desired.

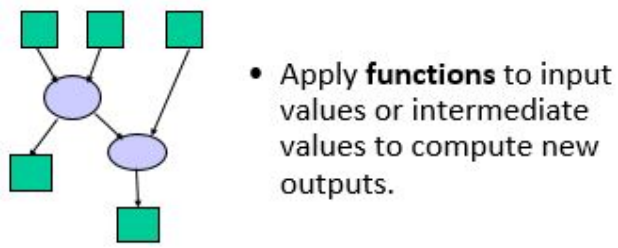
This simple example provides a clear illustration of how imperative code focuses on describing the steps involved in a computation, rather than simply defining the result. Every intermediate step, such as initializing variables and updating their values through iterations, is stated explicitly: this provides precise control over the computational process, but also underscores how the code serves more as a recipe to achieve the result rather than a description of the result itself. In this particular example, the loop specifies each individual step required to calculate the factorial of the given number, emphasizing the *how* to reach the result, as opposed to the *what* the result is.

A substantial body of the existing languages are imperative, including popular languages such as C++ and Java.

> **Side remark**. One may argue that C++ and Java are, in fact, object-oriented languages, and hence should be categorized accordingly. That is a fairly common approach in textbooks and in the academic literature, but we should bear in mind that object-orientation, and similarly, concurrency, are second-tier aspects in programming language design with respect to the fundamental branching between imperative and non-imperative (declarative).
> Object orientation, offers mechanisms for data abstraction, by providing a uniform access mechanism to data through the methods (procedures or functions) collected in the interface that the data advertises to its clients. Which such mechanism have proven effective to organize large programs and code bases, they are fully amenable to implementations in imperative languages (as in Java) as well as in functional languages (as, e.g. in Scala). We'll return to that shortly.

Functional programming

In contrast to imperative languages, computation in **functional languages** is the result of function application and function composition. **Programs** are constructed as **expressions** representing functions that are applied to the data they receive as input and compute values that they return as the result of their evaluation.



As a result, **computations** in functional languages are **described very naturally by substitution** (of the formal function parameters with the actual arguments received as input) **and rewriting**, without any

reference to any implicit state or other underlying computational structure. Function definitions can be as abstract as their **pure mathematical definitions.**

For example, here is the most direct definition of the factorial function, which is indeed a valid functional (Scala) program:

```
def fact (n: Int) : Int =  if (n <= 1) then 1 else n * fact(n-1)
```

State-oriented computations may still be accounted for with functions, but accomplished by carrying the state around explicitly rather than implicitly. For example, a more efficient (yet, certainly less intuitive) functional implementation of the factorial may be defined as follows:

```
def fact(n: Int) =
     def factorial(n: Int, acc: Int): Int =
           if (n <= 1) acc else factorial(n - 1, n * acc)
     factorial(n, 1)
```

The formal parameters `n` and `acc` are examples of carrying the state around explicitly, and the recursive structure has been arranged so as to mimic as closely as possible the looping behavior of the program given earlier. Note that the conditional in this program is an expression rather than command; that is, it denotes a value (conditional on the value of the predicate) rather than a sequencer of commands. The value of the program is the desired factorial, rather than it being found in an implicit store.

Functional languages and functional programming *style*

Functional languages are characterized by few **key distinguishing features**:

- **higher-order functions** (and, sometimes, lazy evaluation)
- **pattern matching**,
- various kinds of **data abstraction** and **polymorphism**

The combination of these features is what qualifies what is commonly referred to as **functional programming style**, which we may define as a programming practice in which such features are manifest, and used extensively, whereas **side effects** are **strongly discouraged but not necessarily eliminated**.

The decision about including or excluding side effects has very significant consequences on modularity and on our ability to reason about program behavior: we'll return to that below, in the closing section on

referential transparency. That said, adopting a functional programming style (whether including side effects or not) has *per se* rather interesting consequences.

To make the point, we look at two examples.

From imperative to functional style (1/2)

We first look at ***iteration***, a powerful abstraction used to **traverse container objects** such as lists, arrays and more generally, collections.

In object oriented contexts, iterators are typically realized as objects associated with a collection, and provide two methods to support the traversal of the collection.

To illustrate, consider **a collection of people from which we want to extract the age of the oldest male**: the example is instructive as it also shows ho OO iterators may be implemented in imperative as well as functional settings.

Our **first implementation is in Java**. Assume we are given a collection `people: Collection<Person>`, where `Person` is a type providing two `get` methods to inspect the attributes `Gender` and `Age`.

The following code snippet computes maximal age of the males inside the collection.

```
// Create a new iterator for the collection
Iterator<Person> it = people.iterator();
int maxAge = -1;
// iterate over the collection
while(it.hasNext()) {
     Person p = it.next();
     if (p.getGender() == MALE && p.getAge() > maxAge)
     maxAge = p.getAge();
}
```

This iteration is expressed in what we may define object-oriented imperative style: object-oriented in the use of the iterator, imperative in the structure of the loop that leaves the result in the variable `maxAge`.

Since **Java5**, the code can be rephrased to make it more elegant, hiding the explicit use of the iterator by resorting to the `for-each` construct available on collections:

```
int maxAge = -1;
for(Person p : people)
```

```
    if(p.getGender() == MALE && p.getAge() > maxAge)
         maxAge = p.getAge();
```

With **Java8** (similarly, in Scala) iteration can be rendered in functional style, by resorting to higher-order methods:

```
final int maxAge = people.stream()
                         .filter(p -> p.getGender() == MALE)
                         .mapToInt(p -> p.getAge())
                         .max();
```

Here the data structure `Collection` is used to produce a stream of elements that proceed through a pipeline of aggregate operations: the first isolates the stream of males, the second creates a stream of ages from which the third collects the max element.

You may **at first find this kind of code difficult to read** because the notation is compact and the flow of control implicit. That is because when formulating algorithms we are used to think in terms of *how* they should compute the intended result more than in terms of *what* they compute.

A comparison between the last two code snippets is illuminating:

- the imperative implementation is a perfect illustration of how to proceed with the calculation, which however leaves it to us to be convinced that the result is indeed what we expect to compute.
- On the other hand, the functional implementation above is a direct, and crystal clear representation of what we wish to compute.

Interestingly, clarity comes at no expense in terms of efficiency: lazy evaluation, available with Java streams, ensures that the final result is computed with just one traversal of the stream.

Furthermore, using `parallelStream()` in place of `stream()` would enable a parallel evaluation scheme in which the Java runtime partitions the stream into multiple sub-streams and let the aggregate operations process these sub-streams independently of each other by minimizing the synchronizations among the parallel threads.

The same would not be possible with the imperative implementation, as in that case the parallel threads would have to synchronize on the updates to the shared memory location `maxAge` to ensure the correctness of the final result.

From imperative to functional style (2/2)

Consider the following while loop for printing the elements of an array of strings.

```
def printArgs(args: Array[String]): Unit =
    var i = 0
    while (i < args.length)
      println(args(i))
      i += 1
```

By no mistake, this is a paradigmatic example of imperative style. An equivalent version of the program, exhibiting some functional flavor is obtained by getting rid of the `var` declaration, for example, as follows:

```
  def printArgs(args: Array[String]): Unit =
    for (arg <- args) println(arg)
```

The refactored code is clearer, more concise, and less error-prone than the original (purely imperative) code. Still, the refactored `printArgs` is not *purely* functional, because it has side effects —in this case, printing to the standard output stream — as it is signaled by the return type `Unit` : if a function isn't returning any interesting value, which is what a result type of `Unit` means, the only way that function can make a difference in the world is through some kind of side effect.

A purely functional version of the code would define a method that formats its arguments to make them ready for printing, and only then proceed with the actual printing, so as to isolate the purely functional components from the components with side effects:

```
def printArgs(args: Array[String]): Unit =
    def formatArgs(args: Array[String]) : String = args.mkString("\n")
        println(formatArgs(args)
```

The `mkString` method, which is available on collections, returns a string consisting of the result of calling `toString` on each element, separated by the string received as argument. Thus if `args` contains three elements `"zero"` , `"one"` , and `"two"` , `formatArgs` will return `"zero\none\ntwo"` : this doesn't actually print anything out, but that is easily accomplished by passing its result to `println` .

The benefits of Functional Programming - Referential Transparency and Equational Reasoning

The benefits of a functional programming *style* (in which side effects are strongly discouraged but not necessarily eliminated) has been advocated in many modern languages: ML and Scheme are the most

notable examples.

On the other hand, there is a very large constituency of *purists* in the functional programming community who believe that purely functional languages are not just sufficient for general computing needs but, better because of their “purity”, as purity brings with it *referential transparency*.

Referential transparency (RT) is often referred to as the quintessence of *pure* functional programing. It is a property of expressions, not just of functions, that we may express as follows:

> *An expression `e` is referentially transparent if for any program `p` containing `e` all occurrences of `e` be replaced by the result of its evaluation without changing the meaning of `p`*

A consequence of this definition is that

> if a variable is bound to a referentially transparent expression, replacing each variable occurrence with its referent expression does not change the result of the computation

Referential transparency is interesting because it forces the invariant that everything that a function *does* is represented by the *value* the function returns.

This invariant also enables a simple and **very natural way of reasoning about program evaluation and behavior known as the *substitution model*.** When expressions are referentially transparent, we can imagine that the **computation proceeds as when we solve an equation**, by expanding all variables with the expressions they are bound to and then reduce the expression under consideration to its simplest form.

As we shall see along the course, this is precisely the way we will formalize the notion of evaluation in our foundational presentation of the theory of programming languages.

Here, we exemplify the effects of RT, and of lack thereof, with two (Scala) examples.

First consider the case of an identifier `x` bound to a string, say `abc`. In Scala, strings are immutable, hence any computation manipulating a string creates a new string leaving the old string unchanged. As a result, an equality test like

```
x.reverse == x.reverse // true
```

will always succeed, as it will the result of substituting `x` with its referent `abc`.

```
"abc".reverse == "abc".reverse // true
```

Now take a new identifier `y` and assume it is initialized to the value `new StringBuilder("abc")`. `StringBuilder` are **much like strings, but certain methods operate on them in place**: one such method is `append`. Thus, unlike the previous case, the test below fails, in spite of the two expressions looking the exact same:

```
y.append("d") == y.append("d") // false
```

Notice that if we replace `y` with its referent expression the result changes:

```
"abc".append("d") == y.append("d") // true
"abc".append("d") == "abc".append("d") // true
```

The **benefits of referential transparency** should already be evident from these examples.

Reasoning on RT expressions is simple and direct as the **effects of evaluation are purely local** (they affect only the expression being evaluated) and we need not mentally simulate sequences of stated updates to understand a block of code. Evaluation is simply simulated by substitution and local rewriting.

As a result, RT **programs are inherently modular**, as they consist of component functions and expressions that can be understood and reused independently of the context, so that the **meaning of their composition only depends on their meaning of the components** and on the rule governing their composition.
Remarkably, this kind of **modular** understanding is also reflected at the **implementation** level, enabling independent (i.e. parallel) evaluation and other very effective optimization techniques.

On the other hand, reasoning on the behavior of non RT expressions inevitably requires to take the context and the order of evaluation into account.

This is not to say that a formal semantics of imperative (or more generally non RT) programs is impossible to achieve (there are, in fact, successful examples of denotational or axiomatic semantic characterizations of such programs), but the direct flavor of equational reasoning is incomparably simpler and more natural.

Is studying Functional Programming a good idea?

As you'll have most probably realized by now, functional programming will play a central role in this course. If you are not yet convinced that studying functional languages is a good idea, here are three final thoughts for you to consider:

Functional programming will make you think differently about programming:

- Mainstream languages are all about state
- Functional programming is all about values

Functional programming is "cutting edge"

- A lot of current research is done based on functional programming
- Rise of multi-core, parallel programming is likely to make minimizing state much more important

New ideas help make you a better programmer, in any language.

CREDITS

The iterator example is based on this paper by Silvia Crafa on an *evolutionary view of programming languages abstractions*.

The section on referential transparency is adapted from the first chapter of Chiusano & Bjanason's textbook on *Functional Programming in Scala*.

FURTHER READING

- B. Goldberg: Functional Programming Languages. ACM Computing surveys, Vol. 28, No. 1, March 1996.
- Paul Hudak: Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, Vol. 14, No. 3, September 1989