The Art of Functional Programming

with examples in OCaml, Haskell, and Java

Minh Quang Tran, PhD

The Art of Functional Programming

About the author:

Minh Quang Tran has over 20 years of experience studying software development and working in the software industry. He has worked in various tech startups and large software companies in Europe. He holds a Bachelor of Computer Science from Furtwangen University, Germany, an M.Sc. in Computer Science from McMaster University, Canada, and a Ph.D. in Computer Science from the Technical University of Berlin, Germany. His interest lies in understanding and mastering the fundamentals and principles that transcend programming languages, frameworks, and tools.

multi-argument function and decide how to arrange the arguments. Currying and partial application are at the core of functional programming languages. Non-functional programming languages like C or Java do not support these techniques.

3.4 General Computation Methods as Higher-Order Functions

So far, we've mainly defined functions whose arguments or return values are numbers, boolean values, or strings. But recall that functions are first-class citizens in functional programming languages. Thanks to this, we can easily define a function that accepts other functions as arguments or returns another function as a result. Such a function is called a **higher-order function**. High-order functions are powerful because they enable us to formulate computation patterns that work with different functions.

3.4.1 Summation as a higher-order function

A good way to appreciate the power of functions operating on other functions is to look at summation in mathematics. For instance, mathematicians often study summations, sums of a sequence of numbers, like the sum of all natural numbers from 1 to n:

$1 + 2 + 3 + ... + n$

Or the sum of squares of natural numbers from 1 to n:

$1^2 + 2^2 + 3^2 + ... + n^2$

If we look at these sums of sequences, we can notice that summation can be defined for any function capable of producing the terms for the sum. Of course, mathematicians realized this a long time ago. They invented the summation symbol $\sum$ (read "sigma") to express the sum of elements represented by a function f within an interval, $[m, n]$.

$\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + ... + f(n)$

The following diagram illustrates how the concept of summation is a generalization of more concrete sum concepts:

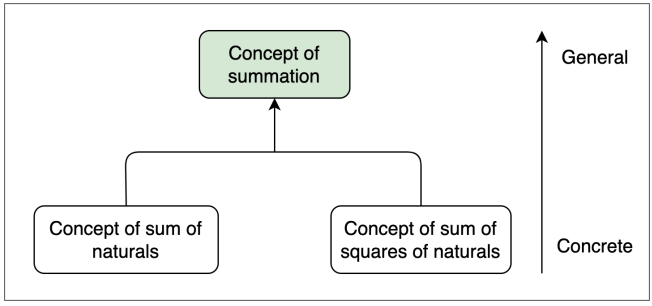

Figure 35: Mathematical summation as generalized concept

What makes $\sum$ so powerful is that it allows mathematicians to think about summation as a concept itself rather than as just the sum of a particular function.

Functional programming readily provides us with a similar power. Let's formulate a higher-order function `sum` in OCaml that behaves similar to the sigma notation $\sum$ above. In particular, it accepts three arguments – a function `term` that maps each index `i` to a term of the

sum, the lower index bound `m`, and the upper index bound `n`. For simplicity, we only consider `term` functions that produce terms of type integers.

```
(* OCaml *)
let rec sum term m n = if m > n then 0 else
    term m + sum term (m + 1) n
```

We can easily formulate a function, `sum_integers`, that sums up all integers in a given range with `sum`. In particular, we pass the identity function to `sum`.

```
(* OCaml *)
let sum_integers m n = sum (fun i -> i) m n

sum_integers 1 3
(* Result: 6 *)
```

Likewise, the function that calculates the sum of squares of integers within an interval is a particular case of `sum`.

```
(* OCaml *)
let sum_integer_squares m n = sum (fun i -> i
    * i) m n

sum_integer_squares 1 3
(* Result: 14 *)
```

What is the difference between the mathematical sigma $\sum$ and our OCaml `sum` function?

$\sum$ represents a mathematical concept of summation. However, our `sum` function encapsulates a computation or algorithm that describes how to actually calculate summation. The following

figure illustrates this:

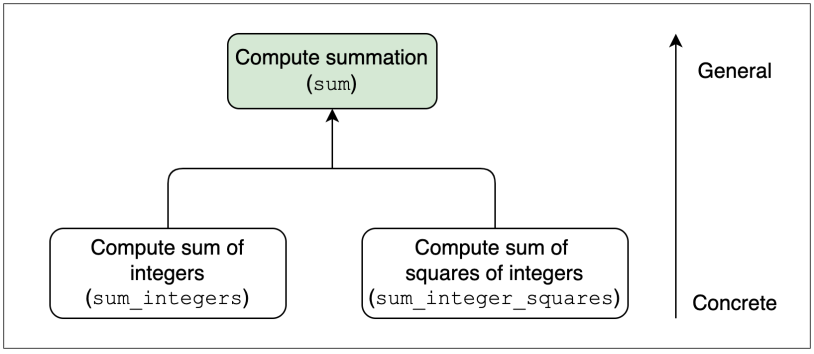

Figure 36: Higher-order function as general method of computation

Although `sum_integers` and `sum_integer_squares` work as expected, their definitions look quite cumbersome. For instance, let's look at `sum_integers` one more time:

```
1  (* OCaml *)
2  let sum_integers m n = sum (fun i -> i) m n;;
```

The definition is verbose because the lower and upper bound, `m` and `n`, are passed unchanged into `sum`. Can we get rid of these arguments?

The answer is yes, we can.

```
1  (* OCaml *)
2  let sum_integers = sum (fun i -> i)
```

This shorter version works because `sum (fun i -> i)` is a partial function application whose result is a function. We can think of this

function as a specialized version of `sum` where we fix the `term` function to become `(fun i -> i)`. Moreover, since the function accepts two arguments, it precisely defines `sum_integers`.

Similarly, `sum_squares` can be defined more clearly as follows:

```
(* OCaml *)
let sum_squares = sum (fun i -> i * i)
```

3.4.2 Accumulation as a higher-order function

We can go even further and treat summation as a particular case of accumulation! To illustrate, let's start by observing that mathematicians also think about the products of sequences of numbers. For instance, the factorial of n, denoted by $n!$ is defined as follows:

$$n! = 1 \times 2 \times 3 \times ... \times n$$

Or the product of squares of natural numbers from 1 to n:

$$1^2 \times 2^2 \times 3^2 \times ... \times n^2$$

Similar to `sum` discussed previously, we can also define a higher-order function `product` that captures the concept of the product for any function.

```
(* OCaml *)
let rec product term m n = if m > n then 1
    else term m * product term (m + 1) n;;
```

We use `product` to define a function `product_integers` that calculates the product of integers within an interval.

```
(* OCaml *)
let product_integers = product (fun x -> x)

product_integers 1 3
(* Result: 6 *)
```

Likewise, we can reuse `product` to formulate a function that calculates the product of the square of integers within an interval.

```
(* OCaml *)
let product_integer_squares = product (fun x
    -> x * x)

product_integer_squares 1 3
(* Result: 36 *)
```

So far, so good. Yet, `sum` and `product` share a common pattern – both accumulate terms produced by a given function within an interval. This means we can factor out the common pattern into an even more general higher-order function and call it `accumulate`. It takes a binary function, `combiner`, that combines the current term with the previous accumulation. It also accepts an `init` argument that represents an initial value. The last remaining arguments are a `term` function and a range, `[m,n]`.

```
(* OCaml *)
let rec accumulate combiner init term m n =
            if m > n then init
            else combiner (term m) (accumulate
                combiner init term (m + 1) n)
```

Next, we can formulate `sum` and `product` as a particular case of `accumulate`.

```
(* OCaml *)
let sum = accumulate (+) 0
let product = accumulate ( * ) 1

sum (fun x -> x) 1 4
(* Result: 10 *)
product (fun x -> x) 1 4
(* Result: 24 *)
```

3.4.3 Climbing up the abstraction hierarchy

Let's stop and reflect on what we've done so far. We started out with concrete computations, such as the sum of natural numbers, and the sum of squares of natural numbers. Next, we abstracted them into a more general computation summation. However, we soon realized summation is just a particular case of an even more general computation accumulation. This led us to capture accumulation as a higher-order function, `accumulate`. All this was possible because functions are first-class citizens in functional programming languages and hence can accept other functions as arguments.

We can view this remarkable process as climbing up the abstraction hierarchy. The higher we move up, the more general methods of computation we obtain. The following diagram depicts this:

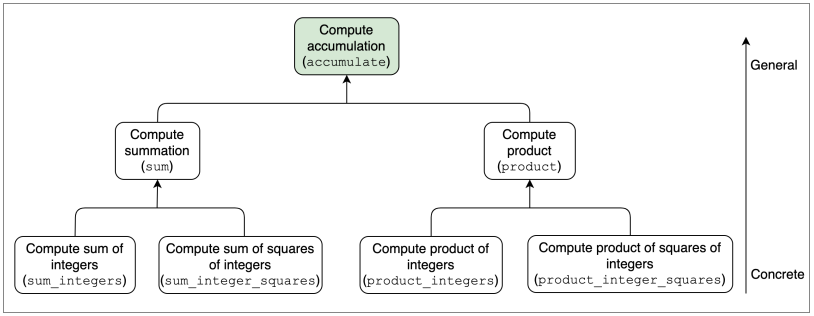

Figure 37: Climbing up the abstraction hierarchy

A general method of computation like `accumulate` has many advantages. First, it avoids code duplication and enables code reusability. In our example, we can reuse `accumulate` to formulate all the functions beneath it in the diagram, such as `sum`, `prod`, `sum_integers`, `prod_integers`, and many others. More importantly, such a general computation method allows us to think about programming on a higher abstraction level. This significantly reduces mental effort when solving programming problems. To see this, let's compare two ways of solving the problem of implementing a function, `sum_integer_cubes n`, that computes $1^3 + 2^3 + ... n^3$.

Low-level thinking

We write a recursive function that returns `0` in case $n \leq 0$. Otherwise, we add the cube of `n` to the recursively calculated result for `n - 1`.

```
1 (* OCaml *)
2 let rec sum_integer_cubes n = if n <= 0 then 0
```

```
        else n * n * n + sum_integer_cubes (n -
       1)
```

High-level thinking

We use the `accumulate` computation pattern where the `combiner` is the addition operator +, the `term` function raises an argument to its cube. Moreover, the initial value, `init`, is 0, whereas the range is `[1, n]`.

```
1 (* OCaml *)
2 let sum_integer_cubes = accumulate (+) 0 (fun
      x -> x * x * x) 1
```

Both approaches arrive at a solution for the problem but the thought process is very different. In the first solution, we are concerned about low-level implementation details, such as handling the different recursion cases. In second solution, we pick `accumulate` from a toolbox and focus primarily on how to configure the parameters of `accumulate` to solve the problem. How `accumulate` is implemented is irrelevant to us at this level of abstraction.

3.5 Recursive Functions

3.5.1 Where are the for/while loops?

If you are new to functional programming, you might be wondering, "How can I make a loop in a functional programming language?" Assume you want to calculate the sum, `1 + 2 + 3 + ... + n`, for

a given `n` natural number. In a non-functional programming language, we typically use a **`for`** or **`while`** loop:

```java
// Java
int sum (int n) {
   int s = 0;
   for (int i = 1; i <= n; i++) {
      s = s + i;
   }
   return s;
}
```

This way of programming is inherently imperative because we explicitly specify the steps required to update `s` via variable assignment within a loop.

In the functional paradigm, however, we express `sum` as an expression. In particular, we can have a mathematical definition of `sum` as follows:

$$sum(n) = \begin{cases} 0, & \text{for } n = 0 \\ n + sum(n-1), & \text{for } n > 0 \end{cases}$$

We can translate this definition directly into a recursive function – a function that calls itself. Such a function is marked with the `rec` keyword in OCaml.

```ocaml
(* OCaml *)
let rec sum n = if n <= 0 then 0 else n + sum
    (n-1)

sum 3
(* Result: 6 *)
```

As you delve into functional programming, your thinking will soon shift from "How can I use a loop to compute this?" to "What is the recursive structure of the computation I'm trying to formulate?". The good news is, many computations in programming have inherently recursive structures, which makes them ideal to be formulated as recursive functions.

As powerful as recursion is, it is associated with an infamous problem – stack overflow. In our case, if we apply `sum` to a large number, the function execution might eventually reach the stack limit. If we calculate `sum 1000000`, we'll see a stack overflow exception.

```
(* OCaml *)
let rec sum n = if n <= 0 then 0 else n + sum
    (n-1)

sum 1000000
(* Result: exception Stack_overflow *)
```

Before discussing solutions for this issue, let's review why the stack overflow problem occurs in the first place. The following visualizes the process evolved from computing `sum 5`.

```
sum 5
5 + sum 4
5 + (4 + sum 3)
5 + (4 + (3 + sum 2))
5 + (4 + (3 + (2 + sum 1)))
5 + (4 + (3 + (2 + (1 + sum 0))))
5 + (4 + (3 + (2 + (1 + 0))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
```