Higher-Order Functions

Axel Månson Lokrantz

Spring Term 2023

Introduction

A higher-order function in mathematics and computer science is a function that either takes a function as an argument or returns a function as its result. All other functions are considered first-order functions. An example of such a function is `map` which takes a function and a collection of elements as argument and as the result returns a new collection where the function is applied to each element from the collection. Other examples are sorting functions which take a comparison function as parameter, `filter`, `fold` and `apply`. The program was written in elixir at KTH.

The task

Apply to all

The first part of the assignment was to understand why it is sometimes beneficial to use higher-order functions by implementing a function `double_five_animal`. The function takes a list of either integers or animals and a second argument. If the second argument is `:double` the program doubles the elements, if it is `:five` we add 5 to all elements, and if the argument is `:animal` we replace all occurrences of `:dog` to `:fido`.

```
def double_five_animal([], _) do [] end
def double_five_animal([h|tail], type) do
  case type do
    :double -> [h * 2 | double_five_animal(tail, type)]
    :five -> [h + 5 | double_five_animal(tail, type)]
    :animal -> [if h == :dog do h = :fido else do h end
    | double_five_animal(tail, type)]
  end
end
```

If we want to add more functionality to our program we need to add more cases. A better solution would be to make a more general solution. If the programming language allows it, a function can be treated as data. in elixir the syntax could look something like this:

```
f = fn(x) -> x + 5 end
```

f is not the name of the function but rather the variable bound to it. To call f we have to use a dot between the function and the parenthesis.

```
f.(10)
```

We can now make a new function apply_to_all that takes a list as its first argument and a function as its second argument.

```
  def apply_to_all([], _) do [] end
  def apply_to_all([h|tail], f) do [f.(h)|apply_to_all([tail], f)] end
```

In this more general approach whatever function we insert will be applied to the list. The function can also be defined directly inside the parameter list when calling apply_to_all.

```
  apply_to_all([1,2,3], fn(x) -> x + 10 end)
```

Reducing a list

We can now implement two functions, fold_right and fold_left. fold_right applies the function in a right to left order. For a list [1,2,3] will get an output that looks like this when paired with a function that adds all the elements in the list. 1+(2+(3+0)).

```
def fold_right([], acc, _) do acc end
def fold_right([h|tail], acc, f) do f.(h, fold_right(tail, acc, f))

def fold_left([], acc, _) do acc end
def fold_left([h|tail], acc, f) do fold_left(tail, f.(h, acc), f)
```

Both functions takes an additional argument which is an accumulator. As acc constantly gets called upon the the result is stored inside accumulator. When we reach our base case, an empty list, the accumulator is returned.

Filter a list

The third higher order construct is a filter function. Implementing a function that filters out all odd numbers with a first-order function would look something like below.

```
def odd([]) do [] end
  def odd([h|tail]) do
    if rem(h, 2) == 1 do
      [h|odd(tail)]
    else do
      odd(tail)
    end
  end
```

Lets say that we now want to create a filter function that can not only filter odd numbers, but also even, all numbers above a certain value etc. Then a more general implementation is preferred.

```
  def filter([], _) do [] end
  def filter([h|tail], f) do
    if filter.(h, f) do
       [h|filter(tail, f)]
    else do
    filter(tail, f)
    end
  end
```