Evaluating an expression

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the way one can evaluate a mathematical expression by making use of the environment implemented last week. Having said that, in order to complete this assignment, the created environment under the form of a list will be used.

The environment

After having defined the literals and the expression forms for the given operations, as far as I am concerned, the **key idea** that stands behind the implementation of this week's assignment is that we are going to use the *add* and *lookup* methods to add in the environment bindings between variables (under the form of an atom) and numerical values, which can be either integer or rational numbers. Ultimately, one will have to search for the information in the environment in order to decide whether it exists or not.

In order to avoid running the commands that compute the mathematical algebra every time from the terminal, a small function has been created in that sense. This function will, therefore, create a new environment (a list), add elements to it (the bindings) and evaluate an expression under the form of an AST.

```
def create_env() do
 env = EnvList.new()
 env = EnvList.add(env, :x, {:q, 12, 4})
 env = EnvList.add(env, :y, {:num, 1})
#eval({:mul, {:num, 5},{:mul, {:var, :x}, {:var, :y}}}, env)
#eval({:div, {:var, :x}, {:var, :y}}, env)
 eval({:add, {:add, {:div, {:num, 2}, {:var, :x}}, {:num, 3}}, {:q, 1,2}}, env)
end
```

Creating the *eval/2* method

Before diving into the approach of implementing this method, it's quite important to note that the task requires to work with *rational numbers*. Therefore, we have to be able to do fraction manipulations with the given operations. Having said that, for generalization purposes, integers will be treated as rational numbers as well, where the denominator will be equal to one *(e.g: {:q, 5, 1})*

The *eval/2* follows the below presented logical steps:

- base case 1: evaluating an integer and turning it into a rational one;
- base case 2: regardless of the environment, any rational number is evaluated to the exact same rational number;
- base case 3: evaluating a variable (the key); if it doesn't exist we display ':error', otherwise we wish to evaluate its associated value by doing the recursive step;

```
def eval({:num, n}, _) do {:q, n, 1} end
def eval({:q, a, b}, _) do {:q, a, b} end
def eval({:var, v}, env) do
  case (EnvList.lookup(env, v)) do
    :nil -> :error
    {_, n} -> eval(n, env)
    #we extract the value from the key and evaluate it using the recursive step
  end
end
```

- a universal construction for the different operations, which only differs in the mathematical approach of obtaining the final expression, which is ultimately simplified using the greatest common divider between the nominator and the denominator;

The universal construction through the 'add' pattern

Evaluating an addition consists in two parts. First, the evaluation through case statements, where we make sure the variables exist in the environment and second, if they do exist, we use them as arguments to call the add method which does the maths and ultimately, calls the simplification method. It should be taken into account that we are always using rational numbers and in order to add two fractions, cross addition will be done.

```
def eval({:add, e1, e2}, env) do
  case(eval(e1, env)) do
    {:q, _, _} ->
      case(eval(e2, env)) do
        {:q, _, _} -> add(eval(e1, env), eval(e2, env))
        :error -> :error
      end
    :error -> :error
  end
end

def add({:q, a, b},{:q, c, d}) do
  simplify({:q, a*d + b*c, b*d})
end
```

As it can be noticed, the implemented cases help us execute the operation only if both arguments exist. Otherwise they return the error atom. One mistake I initially did was that I forgot that the *eval/2* function was returning ':error' if the binding did not exist. Therefore, up above instead of *:error -> :error*, I wrote *:nil -> :error*. Having said that, every time we call a certain function, we have to make sure we keep track exactly what it returns so that we can reuse that information in order to, ultimately get the correct results.

In order to implement *the multiplication, subtraction and division*, the only thing we have to do is naturally, change the atom in the pattern and then, create the *mul, sub and divide* methods that do the maths behind each operation. Therefore, apart from that and depending on the operation, the following will be the arguments used to call the simplify method:

- multiplication: *simplify(:q, a*c, b*d)*
- subtraction: *simplify(:q, a*d - b*c, b*d)*
- division: *simplify(:q, a*d, b*c)*

Note: It should be taken into account that in the above notation, the nominator is under the form of $\frac{a}{b}$ and the denominator under the form of $\frac{c}{d}$.

Conclusion

Looking back, I think that the most difficult part in doing this task was understanding the overall picture of it, namely what an environment is, how we 'feed' it with information and especially that the key-value association is actually the binding between the variable and its numerical value in order to ultimately, do the mathematical calculations.