Evaluating Expressions With Custom Types & Environment

William Frid

Spring 2024

Introduction

Evaluating expressions in Elixir can be done in many different ways. One way that this report focuses on, is by using custom types and an environment holding values of variables. This is a rather simple way of doing this but oh so powerful and clean if done correctly.

The report will start by focusing on the structure of the code, then dive deeper into the decisions made to achieve simplicity, and the somewhat more complex functions (very few). The goal is to have a working list-based environment that has a function to be built and searched through (a lookup function). Plus support for the numbers, variables, and rational numbers. As well as the mathematical operations addition, subtraction, division and multiplication.,

The Environment

The program that's required is built on two files (holding two modules). One is called `Env`, which holds the environment, and `Expression`, which holds all logic for the expression as well as the custom types used (see section "Evaluating Expressions"). The module holding the environment can be built using only two functions, namely `Env.new/1` which generates a new list-based environment based on the given input, and `Env.lookup/2` which looks up the value of a variable inside the given environment. Both these functions are seen below.

```
def new(bindings) do bindings end
def lookup([], _) do "Error..." end
def lookup([{k, v} | h], var) do
    if k == var do v else lookup(h, var) end
end
```

Evaluating Expressions

To handle the evaluation of expressions in the most simple way possible, one can start by converting each variable to a number (trough `Env.lookup/2`), and each number to a rational number. Then perform all the calculations on those rational numbers, and finally simplify and convert to a normal number if possible (for example $\{:num, 5\}$). This method makes it possible to write all functions called `Expression.eval/2` as "one-liners" and only two longer functions (see snippet below).

The function `Expression.evaluate/2` is rather self-explaining, but-`Expression.simplify/1` further down in the snippet is a bit more interesting and worth talking about. The function's goal is to simplify rational numbers. Meaning to make the numerator and denominator as small as possible but remain natural numbers. This is accomplished using modulo. Starting with the value of the denominator, then decreasing by one each recursive call until it finds a suitable value to divide both the numerator and denominator with. Note that it will stop if it reaches the value of one.

Now, assume one have an environment holding the variables $x = 2$, and called `Expression.evaluate(:mul, :num, 2, :div, :num, 3, :num, 4, env)`. The function `Expression.evaluate/2` would start by passing the given data to `Expression.eval/2` which uses function overloading and pattern matching to first convert the values into rationals, then to "dive deeper" into the problem to perform the calculations(simple thanks to all values being rationals). After the calculations of the rational numbers are done, the final rational number is simplified using `Expression.simplify/1` and then converted into a number if the denominator of the rational number is equal to one.

```
def evaluate(e, env) do
    {:q, v1, v2} = simplify(eval(e, env))
    if v2 == 1 do {:num, v1} else e end
end
def simplify({:q, v1, v2}) do simplify({:q, v1, v2}, v2) end
def simplify({:q, v1, v2}, d) do
    if d > 1 and rem(v1, d) == 0 and rem(v2, d) == 0 do
        {:q, v1/d, v2/d}
    else
        if d > 2 do
            simplify({:q, v1, v2}, d - 1)
        else
            {:q, v1, v2}
        end
    end
end
```

Discussion

This approach to solving the problem, that is converting everything into rational numbers, might not yield the fastest results. It does however make it possible to write very simple code which in this case where performance isn't in focus.

In addition, the course of using a list as an environment isn't always the best choice. In this case, where it holds very few variables it's fine. But assume you were to fill it with thousands of variables, then the lookup would be rather slow. In this case, you'd be better off using a binary y tree as the environment instead due to its characteristics (high speeds due to $O(logn)$.

Conclusion

Choosing the correct data structures and methods for tasks similar to this can simplify the code structure and logic drastically. Environments too are important to develop properly as these easily can end up being bottlenecks in the software if done incorrectly.

The approach for evaluating mathematical expressions presented in this report is well suited for its task. It's a clean method that offers great scalability at a low cost, as the extra computations made are done upon converting numbers and variables (with their values) into rational numbers.