Derivative

William Frid

Spring 2024

Introduction

When learning about programming it's usually a good idea to explore different forms of programming languages. Some of which are interpretive, declarative, and object-oriented (OOP). This report will focus on the language known as Elixir. A functional, concurrent, and overall general-purpose language. It covers the creation of a program that can perform simple derivation and handle the following mathematical operations: addition, exponential, multiplication, natural logarithm, division, square root, and sin.

Achieving this gives a deeper understanding of the base structure of Elixir and an improved understanding of functional programming that relies heavily on recursion. As Elixir uses modules, all functions and custom types will be kept in a module called "Derivative". The first section covers the custom types as seen under the section "Custom Types".

Custom Types

The first step will be looking at the custom types. The first type, value ($literal()$), covers the basic numbers and variables (note that a variable with a value is a constant, e.g. $\pi = \{:var, 3.14\}$). Next, we have the overall mathematical expressions ($expr()$) which include $literal()$. These two make up all the main data we'll be using and will be used with comparisons to identify cases (more on this under the section "Finding The Derivative").

```
@type literal() :: {:num, number()}
| {:var, atom()}

@type expr() :: {:add, expr(), expr()}
| {:exp, expr(), expr()}
| {:mul, expr(), expr()}
...
| {:sin, expr()}
| {:cos, expr()}
| literal()
```

Finding The Derivative

Now, how does one go about finding the derivative using recursion? The way this was done for this report was by using multiple functions sharing the same name but with different inputs (called "function overloading") to catch each possible case. Each function takes in a unique type of data ($expr()$) that is matched to different cases. Then depending on the case, the function $deriv/?$ calls upon itself but with different data, so a different version of it will catch the call. This goes deeper and deeper (as recursion does) until the most basic derivation functions are reached. Those functions are the ones handling numbers, constants, and variables.

The code snippet below is a clear example of this where the first function finds the derivative of a constant ($\frac{d}{dx}c \equiv 0$ where c is a constant) and the second function covers the derivative of an natural logarithm ($\frac{d}{dx}\ln x \equiv \frac{1}{x}$).

```
def deriv({:num, _}, _) do {:num, 0} end

def deriv({:ln, e1}, v) do
    case e1 do
        {:num, _} -> {:num, 0}
        {:var, _} -> {:exp, e1, {:num, -1}}
        _ -> {:mul, {:exp, e1, {:num, -1}}, deriv(e1, v)}
    end
end
```

For example, assume we start by sending the following data to $deriv/?$: $\{: ln, \{: mul, \{: num 42\}, \{: var, : x\}\}\} = \ln 42x$. This is caught by the version of the function handling the natural logarithm cases. It detects that the first what's inside is of type multiplication, so it sends that forward (recursion) to a different function sharing the same name. Now, the second version returns the derivative which is 42 to the first function. Then the first function, by using basic math rules, generates the expression $\frac{1}{42x} * 42$. And that's it!

Reusing Functions

As some math problems can be rewritten in terms of other operators one can reuse some of the $deriv$ functions. One of those cases is when searching for the derivative of a fraction. The quotient rule claims that $\frac{d}{dx}[\frac{f(x)}{g(x)}] \equiv \frac{g(x)f'(x)-f(x)g'(x)}{(g(x))^2}$, which is equivalent to $(g(x)f'(x) - f(x)g'(x)) * ((g(x))^{-2})$. What is done is that we convert a fraction problem into a multiplication and exponential problem, and by doing so, reusing code that's already been written. See the snippet below.

```
def deriv({:div, a1, a2}, v) do
    {:mul, {
        :add,
        {:mul, a2, deriv(a1, v)},
        {:mul, {:mul, a1, deriv(a2, v)}, {:num, -1}}
    }, {:exp, a2, {:num, -2}}}
end
```

Simplification

Assume the function $deriv/?$ returns something that looks like this: $\{: add, \{: mul, \{: num, 1\}, \{: var, : x\}\}, \{: num, 0\}\} = 1x + 0$ It's clear that this can be simplified to just x. But how? The answer is simple and shockingly similar to what was used for finding the derivative itself (but a bit more chunky).

What was done in this case was the creation of the function $simplify/?$. Once again, one name, multiple versions with different parameters. The goal with these is to perform simple checks and return either just simplifications of the inputs or the value zero ($\{: num, 0\}$). This also goes as deep as possible using recursion until the maxing depth is reached, followed by the values being returned down the stack (a good idea is to see the recursion queue as a stack). See the example below.

```
def simplify({:var, v}) do {:var, v} end

def simplify({:add, e1, e2}) do
    case [simplify(e1), simplify(e2)] do
        [{:num, 0}, {:num, 0}] -> {:num, 0}
        [_, {:num, 0}] -> simplify(e1)
        [{:num, 0}, _] -> simplify(e2)
        [{:num, v1}, {:num, v2}] -> {:num, v1 + v2}
        _ -> {:add, simplify(e1), simplify(e2)}
    end
end
```

But what if we could choose the final structure? For instance, let's assume we have the two options $x(y+2)$ and $xy+2x$. By using the custom types declared, the first option would require one multiplication block and one addition block, while the second one would require two multiplication blocks and one addition block. Comparing these two scenarios it's obvious that the first option is indeed the simpler and cleaner option. It would be referred to the simplified version in mathematical terms. This, however, cannot always to archived by this code as it depends on how the data was structured upon input and the found derivative.

Prettify For Easier Testing

The last function used was one called *prettify/?*. By using recursion it replaces the custom data types (see section "Custom Types") with normal math symbols. Converting data like this helps debugging when working with problems like this. As this function was very simple and the idea builds on the same as for *derive/?* and *simplify/?*, I'll just paste a short snippet below for you as a reader to understand.

```
def prettify({:var, v}) do v end
def prettify({:num, n}) do n end
def prettify({:ln, a1}) do "ln(#{prettify(a1)})" end
```

Discussion

A different approach for the function *derive/?*, instead of using the control flow structure "case", would have been to create more functions where the exact case would have been identified by matching the arguments to the parameters. A slightly cleaner approach might be better suited for functional programming as it gives a simpler flow to debug.

Even though the function *simplify/?* does indeed simplify the output drastically, it doesn't do it perfectly because it cannot read between depths (due to the recursion nature). An example of this (with the data prettified via *prettify/?*) is the expression $((2*(5*x))*5)$. This cannot be shortened by the code written but is doable. To the recursive program, this is impossible to fix, but for us humans, it's $50x$. This might not seem like a big problem, but now imagine this with longer expressions and complex outputs with more types of math operations.

Conclusion

In conclusion, functional and recursive programming can be very effective methods that offer a different form of workflow. Tools such as functions to clarify data and the build tool Mix, which is used for Elixir, can be used at great advantage. This combined with clear documentation is important when working with a language like Elixir due to its different nature (compared to Java, Python, JavaScript, etc.) which can be trickier than some others.