Taking the Derivative

Seema Fatima Bashir

19-01-24

Introduction

The calculations of derivatives hold great importance in mathematics. Those familiar with manual derivation understand that the final result can only be accurately assessed once the function has been simplified to its fullest extent. A simplified expression not only enhances clarity but also ensures smoother application when substituting numerical values into the function.

This task involves the creation and evaluation of derivative rules implemented in the Elixir programming language. Additionally, it focuses on devising rules to simplify the derived expressions and concludes with the printing of these expressions in their most simplified form.

Finally, a short discussion will assess the concept of the simplest form in the context of Elixir. It will explore the ease of achieving simplicity, the potential challenges posed by Elixir, and the significance of having expressions in their simplest form in mathematics. The discussion will highlight practical scenarios, such as the application of a calculation function, where the simplest form becomes essential.

Derivative Rules

Deriving an expression

The process of deriving an expression is initialized by defining the various @types including literals and expressions to represent the different mathematical expressions. Thereby, a series of derivative rules are implemented which can be used to perform the derivative on a given expression, which itself can be a combination of variables (atomic values) and numbers. The first task consists of taking the derivative of the following expression:

```
e = {:add, {:add, {:mul, {:num, 2}, {:exp, {:var, x}, {:num, 2}}},
{:mul, {:num, 3}, {:var, x}}}, {:num, 5}}
```

The various derivative rules used to take the derivative of this expression cover scenarios such as handling constants, variables, addition, multiplication, and the power rule for expressions with variables raised to a constant power. The clauses specify that the derivative of a constant is always zero, a variable's derivative with respect to itself is one, and the derivative of a constant variable is zero. Addition and multiplication rules involve the recursive application of the derivative to individual terms, while the power rule simplifies derivatives of variables raised to a constant power.

Additionally, a print function is implemented for refining the printing of mathematical expressions. Various clauses are created in-order to construct visually appealing string representations of the corresponding mathematical expressions. The patterns used in these clauses follow common notation, ensuring that the printed output is readable.

Furthermore, these clauses follow common mathematical notation, ensuring that the printed output is both readable and familiar to users. Thus providing a clear and concise way to display complex mathematical expressions for better understanding and interpretation.

```
Expression: (((2 * x^2) + (3 * x)) + 5)
Derivative: ((((0 * x^2) + (2 * (2 * x^1))) + ((0 * x) + (3 * 1))) + 0)
```

The visualization above illustrates the outcome after computing the derivative of the given expression. It can be noticed that this answer is filled with syntax, which can be further simplified by applying mathematical rules, offering a more concise solution. Thus, simplification clauses known as simplify/1 are created to transform mathematical expressions into their simplest and most comprehensive form. Each clause caters to specific mathematical operations, such as addition, multiplication, exponentiation, logarithms, square roots, and negation, recursively simplifying the input until optimal reduction is achieved. Thereby, after the implementation of the simplified, the derivative of the expression is visualized in its simplest form.

```
Simplified: ((4 * x) + 3)
```

Additionally, among the simplify/1 clauses, there are also private simplify/2 functions. These private functions play a crucial role in helping further simplify expressions. For instance, simplify-add/2 focuses on simplifying addition operations, eliminating zero terms, and combining like terms. Similarly, (simplify-mul/2) streamlines multiplication expressions by handling cases involving zero, one, and common variable factors. The simplify-exp/2 clause tackles exponentiation, reducing expressions involving powers or constants. Additionally, simplify-ln/2 addresses logarithmic functions, condensing them to more concise forms. These specialized rules collectively

contribute to the overall simplification process by systematically handling distinct mathematical operations.

Expanding Deriv/2 Rules

0.1 Derivative of ln(x)

The logarithm rule (derive/2 clause for ln) is expressed using:

```
def deriv({:ln, e}, v) do {:div, deriv(e, v), e} end
def derive ({:ln, {:num, _}}, _) do {:num, 0} end
```

This rule is designed to find the derivative of the natural logarithm of a given expression. It encapsulates the logic for computing the derivative of the natural logarithm of an expression represented by e with respect to the variable v. Following the chain rule, it recursively calculates the derivative of the inner expression e using the same deriv/2 function and constructs the result as a tuple with the numerator being the derivative of e and the denominator being the original expression e.

The second function, `def derive({:ln, {:num, _, _) do {:num, 0} end`, addresses scenarios where the natural logarithm is applied to a numerical constant. In such cases, the derivative of ln(constant) is always zero. This clause optimizes the computation by providing a predefined result, `{:num, 0}`, when encountering the derivative of a constant value under the natural logarithm. It contributes to the simplification of expressions by recognizing and handling specific cases during the derivative computation process.

0.2 Derivative of $\sqrt{x}$

The square root rule ($\sqrt{x}$) is expressed using the following clauses:

```
def derive({:sqrt, e}, v) do
    {:mul, {:div, {:num, 1}, {:mul, {:num, 2}, {:sqrt, e}}},
    derive(e, v)}
  end
def derive({:sqrt, {:num, _}}, _) do {:num, 0} end
```

The first clause is created in-order to calculate the derivative of the square root of an expression represented by e with respect to the variable v. This derivative clause is tested with the following expression: $\sqrt{2x}$.

```
Expression: sqrt((2 * x))
Derivative: (1 / (2 * sqrt((2 * x))) * ((0 * x) + (2 * 1)))
Simplified: (1 / (2 * sqrt((2 * x))) * 2)
```

In the given expression, sqrt((2 * x)), the goal is to calculate the derivative with respect to the variable x. The derivative is computed as per the chain rule. The derivative of the outer square root function is multiplied by the derivative of the inner function, which is 2 with respect to x. To simplify this expression, it is reduced to (1 / (2 * sqrt((2 * x))) * 2), recognizing that the term (0 * x) contributes nothing to the overall result. The simplified form emphasizes efficiency by eliminating unnecessary terms, providing a more concise representation of the derivative. It can however be noted that the simplified derivative expression (1 / (2 * sqrt((2 * x))) * 2) can be further refined by canceling out the common factor of 2 in the numerator and denominator using more comprehensive division rules.

0.3 Derivative of sin(x)

The Elixir rule `def derive({:sin, e}, v) do {:mul, derive(e, v), {:cos, e}} end` is designed to calculate the derivative of the sine function (sin) applied to an expression e with respect to the variable v. This rule implements the chain rule of differentiation by combining two components: the derivative of the inner function e with respect to v (derive(e, v)) and the cosine of the original expression e ($\{:cos, e\}$), which represents the derivative of $\sin(e)$ concerning e. The multiplication of these components yields the derivative of $\sin(e)$ with respect to v.

To validate the functionality of this rule, a specific test case is executed:

```
def test_inverse_sine_expression do
    x = :x
    e = {:div, {:num, 1}, {:sin, {:mul, {:num, 2}, {:var, x}}}}
    d = derive(e, x)
    IO.puts("Expression: #{pprint(e)}")
    IO.puts("Derivative: #{pprint(d)}")
    IO.puts("Simplified: #{pprint(simplify(d))}")
  end
```

In this test, the expression $e = \frac{1}{\sin(2x)}$ is chosen, and the derivative of e with respect to x is calculated using the implemented rule. The results, including the original expression, the calculated derivative, and the simplified form of the derivative, are then displayed. The answer indeed yeilds the correct derivative:

$$-\frac{2\cos(2x)}{\sin^2(2x)}$$

.

Discussion

The exploration of achieving the simplest form in Elixir reveals both simplicity and potential challenges in transforming mathematical expressions. While Elixir allows for straightforward transformations, the problem arises in determining the most simplified expression. The significance of simplicity, highlighted in practical scenarios like calculation functions, underscores the importance of refining expressions to their most straightforward and clear representation.