Taking the Derivative

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

You of course know how to take the derivative of a function, this is something you probably learned already in high school. There are some simple rules that one applies and in five minutes one can find the derivative of even complicated functions.

Our task in this tutorial is to work with functions as symbolic expressions and compute the derivative of functions. Part of the problem is to understand how we can represent functions using the data structures that we have in Elixir. Once we know how to do this the derivation will be quite simple.

1 Representing functions

You might wonder what the problem is; after all we represent a function in Elixir using the syntax:

```
fn(x) -> x * 2 end
```

This is only partly true, what we do is express what the function should look like, how it is actually represented is hidden to us. If we write:

```
f = fn(x) -> x * 2 end
```

then `f` is a function of one argument that we can use, but if we only have access to `f` we can not determine that it is a function of one argument nor that the the body of the function is a multiplication of the argument and a constant.

In order to differentiate the function `f`, with respect to `x`, we need to know that the body of the expression is `x * 2`. We need to be able to examine the expressions `x*2`, and determine that it is a product of the variable `x` and `2`, only then can we determine that the derivative is 2. So, just because we have *functions* in Elixir, does not mean that we necessarily should use them to represent our "functions".

2 An expression

Let's start by finding a representation for constants and variables. We can limit ourselves to the domain of reals and a real number could of course be represented by an Elixir number. This is of course trivial but remember to separate the real number 2.34 from the representation in Elixir by `2.34`. A variable could be represented as an atom so the variable x is represented by the Elixir atom `:x`.

This simple mapping of elements in our domain to Elixir data structures is tempting to use but it has some disadvantages, we can not use pattern matching to determine if an element is a number or a variable. In our program we would have to use the built-in recognizers to separate the two cases; we would use code as the following:

```
def derivative(n) when is_number(n) do: ...

def derivative(n) when is_atom(n) do: ...
```

We would also have to think twice when we want to include constants such as π, could we represent it using the atom `:pi`? Moreover, when we derivatives, is it important to separate the constant π from the constant 2.34?

A better approach (all though it will turn our expressions into huge data structures) is to be more explicit in our choice of representation. What if we choose to represent all constants using the tuple {`:const, c`} where `c` could be either an atom (`:pi`) or a number (`2.34`). Variables could, in the same way, be represented by tuples {`:var, v`} where `v` is a atom that identifies the variable.

If constants and variables are our `literals`, we have the following definition:

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

3 Expressions

Assume that we, for the time being, limit ourselves to the arithmetic operations multiplication and addition, we have a very natural representation. We will of course represent them using tuples where the first element is an atom that identifies the operation. We have {`:mul, a, b`} and {`:add, a, b`}. Expressions are thus:

```
@type expr() :: {:add, expr(), expr()}
              | {:mul, expr(), expr()}
              | literal()
```

This gives us everything we need to represent a limited sets of expressions. The expression $2x + 3$ could for example be represented by the Elixir structure:

```
{:add, {:mul, {:const, 2}, {:var, :x}}, {:const, 3}}
```

As you see it it is not a syntax we would like to use when we write expressions by had but it has its advantages when it comes to handle the expressions using Elixir clauses.

4 The derivative of

What are the rules of derivation? You of course remember that derivative of $2x + 3$ with respect to x is 2, and that the derivative of x^2 is $2x$ but now we should define a program that does this automatically so we need to have very clear understanding of the rules. If you have not done so yet, this is the point where you should brush up on derivative rules so that you can follow the reasoning.

These are two rules that we will use:

- $\frac{d}{dx}x \equiv 1$
- $\frac{d}{dx}c \equiv 0$ for any literal different from x
- $\frac{d}{dx}f(x) + g(x) \equiv f'(x) + g'(x)$
- $\frac{d}{dx}f(x) \cdot g(x) \equiv f'(x) \cdot g(x) + f(x) \cdot g'(x)$

The third rule is quite straight-forward; the derivative of an sum is the sum of the derivatives of the terms. The derivative of $4x^2 + 2x + 5$ is $8x + 2$ since since the derivative of $4x^2$ is $8x$ etc.

The last rule, you might not even have learned as a rule but simply it's consequences in the most common cases. It is the rule that says that $4x^2$ is $8x$ and $2x$ is 2. When learning using examples like these the general rule is quite simple: multiply the constant with the power and reduce the power by one. The general rule gives us the definition for any product, be it $2x$ or $2x^2$.

5 The rules of the game

So if we know how expressions are represented and how to take the derivative of sums and products, we are ready to implement the rules. This is a skeleton on what a function `deriv/2` would look like:

```
def deriv({:const, _}, _), do: ...

def deriv({:var, v}, v), do: ...

def deriv({:var, y}, _), do: ...

def deriv({:mul, e1, e2}, v), do: ...

def deriv({:add, e1, e2}, v), do: ...
```

What is the derivative of $2x^2 + 3x + 5$ with respect to x? How do we represent the expression in our system? Can you calculate the derivative using the `deriv/2` function?

The answer that you get might not look like the answer you would have hoped for but it might be that what you see is an expression that can be simplified. The derivative of $2x$ is of course 2 but our function will return something that looks like $0 \cdot x + 2 \cdot 1$, which of course is equal to 2.

6 Carrying on

Add more rules to the `deriv/2` function. We should of course be able to take the derivative of the following expressions:

- x^n
- $\ln(x)$
- $\frac{1}{x}$
- $\sqrt{x}$
- $sin(x)$

To be able to handle these expressions you of course need to find a suitable representation. You then have to find the general rule for finding the derivative.

7 Simplification

The results of our derivation might be correct but they are very hard to read. They contain multiplications with zero, addition with constant values etc. All of those could be removed if we simplified the results.

Simplification could be tricky, You could start by transforming an expression so that all functions with constant arguments were actually evaluated. You could then remove expressions that are multiplied with zero etc. The

problem is to know if there are any more things that could be done; how do we know that will not be able to do more.

There will also be a discussion of what the simplest form would look like. Should we write

$$x \cdot (y + 2)$$

or should we write

$$xy + 2x$$