# Learn You a Physics for Great Good!

# >>> WORK IN PROGRESS <<<

## Calculus / Function expressions

[src: Calculus/FunExpr.lhs] Previous: Introduction Table of contents Next: Differential calculus

```haskell
module Calculus.FunExpr (FunExpr (..), RealNum, RealFunc) where

import Test.QuickCheck
```

## Semantics, syntax, and stuff

What is a value in calculus? What kind of values do functions in calculus operate on and produce?

Let's look at derivatives to get an idea of what the semantic value of calculus is.

$$\frac{dx^2}{dx} = 2x$$

$$\frac{df(x)}{dx} = f'(x)$$

Hmm, these examples left me more confused than before. The differentiation function seems to take an expression as an argument, and return the differentiated expression, with regards to a variable. But what is an expression represented as a semantic value? It's not a number yet, the variable in the body needs to be substituted first in order for the expression to be computable. Is it some kind of function then? Well, yes it is! If we reinterpret the differentiation expressions above, it makes more sense.

$$\frac{dx^2}{dx} = 2x$$

can be written as

$$D(x^2) = 2x \text{ with regards to } x$$

which is really equivalent to

$$D(x \mapsto x^2) = x \mapsto 2x$$

or with more descriptive function names

$$D(square) = double$$

.

So the type of unary real functions seems like a great fit for a semantic value for calculus, and it is! Great! But... how do we represent a real number in Haskell? There is no `Real` type to use. Well, for the sake of simplicity we can just say that a real number is basically the same as a `Double`, and really it is (basically). The problem with `Double` is that it's of finite precision, so rounding errors may occur. We'll have to keep that in mind when doing calculations!

```
type RealNum = Double
type RealFunc = RealNum -> RealNum
```

Now, to the syntax. We've concluded that real functions are really what calculus is all about, so let's model them. We create a data type `FunExpr` that will represent symbolic expressions of functions in our language.

`FunExpr` is very expressive!

```
data FunExpr
```

First of all, there are the elementary functions. We can't have them all, that would get too repetitive to implement, but we'll put in all the fun ones.

```
    = Exp | Log | Sin | Cos | Asin | Acos
```

Then, there are the arithmetic operators. "But wait", you say, "Aren't arithmetic operators used to combine expressions, not functions?". We hear you friend, but we will do it anyways. We could make a `Lambda` constructor for "VAR $\mapsto$ EXPR" expressions and define the arithmetic operators for the expression type, but this would make our language much more complicated! Instead, we'll restrain ourselves to single variable expressions, which can be represented as compositions of unary functions, and define the arithmeric operators for the functions instead.

$$f\ OP_{r \to r}\ g = x \mapsto (f(x)\ OP_r\ g(x))$$

```
  | FunExpr :+ FunExpr
  | FunExpr :- FunExpr
  | FunExpr :* FunExpr
  | FunExpr :/ FunExpr
  | FunExpr :^ FunExpr
```

And then there's that single variable. As everything is a function expression, the function that best represents "just a variable" would be $x \mapsto x$, which is the same as the $id$ function.

```
  | Id
```

In a similar vein, the constant function. $const(c) = x \mapsto c$

```
  | Const RealNum
```

Then there's function composition. If you didn't already know it, it's defined as

$$f \circ g = x \mapsto f(g(x))$$

```
  | FunExpr :. FunExpr
```

Finally, the real heroes: the functions of difference, differentiation, and integration! They will be well explored later. But for now, we just define the syntax for them as

```
  | Delta RealNum FunExpr
  | D FunExpr
  | I FunExpr
```

Even more finally, we add a `deriving` modifier to automatically allow for syntactic equality tests between `FunExpr`s.

```
  deriving Eq
```

Nice! This syntax tree will allow us to do symbolically (at the syntax level) what we otherwise would have to do numerically (at the semantics level).

Before we move on, we just have to fix one thing: the operator precedence! If we don't do anything about it, this will happen

```
ghci> Id :+ Id :* Id == (Id :+ Id) :* Id
True
```

Now this is obviously wrong. *Plus* doesn't come before *times*, unless I accidentaly switched timelines in my sleep. To fix this, we have to fix the fixity. `infixl` allows us to make an operator left-associative, and set the precedence.

```
-- Medium precedence
infixl 6 :+
infixl 6 :-
-- High precedence
infixl 7 :*
infixl 7 :/
-- Higher precedence
infixl 8 :^
-- Can't go higher than this precedence
infixl 9 :.
```

## A structure with class

Now that we've defined the basic structure of our language, we can instantiate some useful classes. There are three in particular we care for: `Show`, `Num`, and `Arbitrary`.

Try modifying `FunExpr` to derive `Show`, so that our expressions can be printed.

```
  deriving (Eq, Show)
```

Consider now how GHCi prints out a function expression we create

```
ghci> carAccel = Const 20
ghci> carSpeed = Const 50 :+ carAccel :* Id
ghci> carPosition = Const 10 :+ carSpeed :* Id
ghci> carPosition
Const 10.0 :+ (Const 50.0 :+ Const 20.0 :* Id) :* Id
```

Well that's borderline unreadable. Further, the grokability of a printed expression is very inversely proportional to the size/complexity of the expression, as I'm sure you can imagine.

So if the derived `Show` is bad, we'll just have to make our own `Show`!

```
showFe :: FunExpr -> String
showFe Exp = "exp"
showFe Log = "log"
showFe Sin = "sin"
showFe Cos = "cos"
showFe Asin = "asin"
showFe Acos = "acos"
showFe (f :+ g) = "(" ++ showFe f ++ " + " ++ showFe g ++ ")"
showFe (f :- g) = "(" ++ showFe f ++ " - " ++ showFe g ++ ")"
showFe (f :* g) = "(" ++ showFe f ++ " * " ++ showFe g ++ ")"
showFe (f :/ g) = "(" ++ showFe f ++ " / " ++ showFe g ++ ")"
showFe (f :^ g) = "(" ++ showFe f ++ "^" ++ showFe g ++ ")"
showFe Id = "id"
showFe (Const x) = showReal x
showFe (f :. g) = "(" ++ showFe f ++ " . " ++ showFe g ++ ")"
showFe (Delta h f) = "(delta_" ++ showReal h ++ " " ++ showFe f ++ ")"
showFe (D f) = "(D " ++ showFe f ++ ")"
showFe (I f) = "(I " ++ showFe f ++ ")"

showReal x = if isInt x then show (round x) else show x
  where isInt x = x == fromInteger (round x)


instance Show FunExpr where
  show = showFe
```

Not much to explain here. It's just one way to print our syntax tree in a more readable way. What's interesting is how we can now print our expressions in a much more human friendly way!

```
ghci> carPosition
(10 + ((50 + (20 * id)) * id))
```

Still a bit noisy with all the parentheses, but much better!

Another class we can instantiate for our `FunExpr` is `Num`. This class allows us to make use of the native Haskell functions and operators for numeric operations, instead of writing our own constructors. This sometimes improves the readability of the code.

```
instance Num FunExpr where
  negate e    = Const 0 :- e
  (+)         = (:+)
  (*)         = (:*)
  fromInteger = Const . fromInteger
  abs         = undefined
  signum      = undefined
```

Third but not least, we'll instantiate `Arbitrary`. This class is associated with the testing library *QuickCheck*, and describes how to generate arbitrary values of a type for use when testing logical properties with `quickCheck`. For example, a property function could be formulated that states that the `:*` constructor of `FunExpr` is associative.

The implementation itself is not very interesting. We generate a function expression that tends to contain mostly elementary functions, arithmetic operations, and a generous dose of constants; with a light sprinkle of differences. We won't include derivatives as all elementary functions have elementary derivatives anyways, and integrals may cause cause approximation errors if we have to numerically compute them at evaluation.

```
instance Arbitrary FunExpr where
  arbitrary =
```

`frequency` "chooses one of the given generators, with a weighted random distribution". By assigning probabilities of generating certain functions more often than others, we can restrain the growth of the generated expressions in complexity.

```
      frequency
        [ (10, genElementary)
        , (10, genBinaryOperation)
        , (10, return Id)
        , (20, fmap Const arbitrary)
        , (10, genBinaryApp (:.))
        , (5 , genBinaryApp Delta) ]
    where genElementary = elements [Exp, Log, Sin, Cos, Asin, Acos]
```

```
genBinaryApp op = fmap (\(f, g) -> f `op` g) arbitrary
genBinaryOperation =     elements [(:+), (:-), (:*), (:/), (:^)]
                     >>= genBinaryApp
```

[src: Calculus/FunExpr.lhs] Previous: Introduction Table of contents Next: Differential calculus

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL