Symbolic Differentiation

COMP40009 – Computing Practical 1

30th October – 3rd November 2023

Aims

- To provide experience with using data types and type classes in Haskell.
- To gain familiarity with Haskell's `Maybe` data type
- To introduce the idea of abstract syntax trees and symbolic computation.

Introduction

Symbolic computation refers to the manipulation of data structures representing terms in some well-defined language. In this exercise we will be working with the language of mathematical expressions, but the same ideas can equally be applied to logic, natural language and, of course, programming languages such as Haskell and Java. Indeed, GHC(i) essentially performs symbolic computation on a data structure (the so-called *abstract syntax tree*) representing a Haskell program.

A mathematical *expression* is defined to be either:

- a number
- an identifier, e.g. x, y, z...
- the application of either a *unary* operator (function), here $-$, sin, cos or log etc., or a *binary* operator, here $+$, $\times$ or $/$, to one or two argument expressions respectively.

Note that expressions are naturally recursive: an expression may contain other (sub)expressions. Some examples of expressions involving the variables x and y that conform to this definition are:

1	a number
x	a variable
$-cos\ x$	negation of the cosine of x
$y + ((2 \times x) - 5)$	the sum of two expressions
$(x + 2) \times log(2 \times x)$	the product of two expressions
$(y + 3)/(x \times x)$	the division of one expression by another

Parentheses are used above to show the order in which expressions are to be evaluated, in the usual way.

Expressions such as these can be represented *abstractly* as objects of type `Expr` as follows:

```haskell
data Expr = Val Double | Id String
          | Add Expr Expr | Mul Expr Expr | Div Expr Expr
          | Neg Expr | Sin Expr | Cos Expr | Log Expr
```

As an example, the mathematical expression $log(2 \times sin(x)) - cos(y)$ will be represented by:

```haskell
-- note that subtraction `x - y` is `x + -y`
Add (Log (Mul (Val 2.0) (Sin (Id "x")))) (Neg (Cos (Id "y")))
```

which is an object of type `Expr`. This is a bit of a mouthful, but you'll see later how to simplify the way such expressions can be written and displayed.

Note that an expression containing references to variables, e.g. $y \times cos(x)$ can also be thought of as a *function* of those variables, viz. $f(x, y) = y \times cos(x)$. For this reason, we will use the words "function" and "expression" interchangeably for the purposes of this exercise.

Expression Evaluation

If we specify values for each of the variables in an expression then it can be *evaluated* by interpreting the functions $+, \times, /, sin, cos, log$ in the usual way. All that's needed in addition is a way of mapping variables to values: we can only evaluate $1 + x$ if we know the value of x, for example. This is commonly referred to as an *environment* and can be represented as a list of (variable, value) pairs[1].

Differentiation

Having an internal representation for expressions means that we can also generate an internal representation of the *differential* of that expression with respect to a specified variable. Hence the term *symbolic differentiation*. The rules you need can be found in Figure 1.

The Chain Rule

Recall that the chain rule specifies how to differentiate the *composition* of two functions:

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

so that, for example, setting $u = E$,

$$\frac{d}{dx} log_e(E) = \frac{d}{du} log_e(u) \times \frac{d}{dx} u = \frac{1}{u} \times \frac{d}{dx} u = \frac{\frac{d}{dx}E}{E}$$

for an arbitrary expression E.

Maclaurin Series

Suppose we know the value of a function, f, at two points x_1 and x_2, say, then we can find a line of the form $a_0 + a_1 x$ that passes through both $f(x_1)$ and $f(x_2)$ and can determine a_0 and a_1 by solving the system of equations:

[1] In practice, looking up values in a list is $O(n)$, which is "poor"; instead you would use a map, which would give $O(\log n)$ or amortized $O(1)$ complexity. In practice `Data.Map` is far more appropriate!

$$
\begin{aligned}
\frac{d}{dx}c &= 0\\
\frac{d}{dx}x &= 1\\
\frac{d}{dx}y &= 0, \quad y \neq x\\
\frac{d}{dx}(E_1 + E_2) &= \frac{d}{dx}E_1 + \frac{d}{dx}E_2\\
\frac{d}{dx}(E_1 \times E_2) &= E_1 \times \frac{d}{dx}E_2 + \frac{d}{dx}E_1 \times E_2\\
\frac{d}{dx}(E_1 / E_2) &= \frac{\frac{d}{dx}E_1 \times E_2 - E_1 \times \frac{d}{dx}E_2}{E_2^2}\\
\frac{d}{dx}sin(x) &= cos(x)\\
\frac{d}{dx}cos(x) &= -sin(x)\\
\frac{d}{dx}log_e(x) &= \frac{1}{x}
\end{aligned}
$$

Figure 1: The rules of differentiation, where E_1 and E_2 are expressions and c denotes a constant.

$$
\begin{aligned}
a_0 + a_1 x_1 &= f(x_1)\\
a_0 + a_1 x_2 &= f(x_2)
\end{aligned}
$$

Given three points we can similarly find a parabola of the form $a_0 + a_1 x + a_2 x^2$ that passes through all of them and can solve to determine a_0, a_1 and a_2. In general, given n points we can approximate the function by a polynomial of order n, viz. $\sum_{i=0}^{n} a_i x^i$. Given infinitely many points, i.e. a complete characterisation of the function, we can represent it as a polynomial with infinitely many terms, provided the series converges[2]. To find the coefficients a_i, $i \geq 0$, we repeatedly differentiate f and set $x = 0$:

$$
\begin{aligned}
f(0) &= a_0 &&= 0! \times a_0\\
f'(0) &= a_1 &&= 1! \times a_1\\
f''(0) &= 2a_2 &&= 2! \times a_2\\
f'''(0) &= 6a_3 &&= 3! \times a_3\\
etc.
\end{aligned}
$$

Thus we obtain:

$$
f(x) = \frac{f(0)x^0}{0!} + \frac{f'(0)x^1}{1!} + \frac{f''(0)x^2}{2!} + \frac{f'''(0)x^3}{3!} + \ldots
$$

[2] Not all functions satisfy this property, for example square root and arctan, but we'll only consider "analytic" functions, i.e. functions whose series do converge.

which is called the *Maclaurin series*[3]. By truncating the series after the n^{th} term we end up with an *approximation* to the function f that is said to be of *order n*.

Getting started

As per the previous exercise, you will use the `git` version control system to get the repository with the skeleton files for this exercise and its (incomplete) test suite. You can get your repository with the following (remember to replace the *`username`* with your own username).

```
git clone https://gitlab.doc.ic.ac.uk/lab2324_autumn/haskellcalculus_username.git
```

Alternatively, if you have set up an SSH key and registered it to Gitlab[4], then you can use the following command instead:

```
git clone git@gitlab.doc.ic.ac.uk:lab2324_autumn/haskellcalculus_username.git
```

This is recommended, and will save you a lot of time in the long run!

Project Structure

In practice, it is a good idea to keep things organised into different files. As such, this exercise has a more expanded structure than the previous ones.

```
haskellcalculus_username
├── src
│   ├── Calculus.hs ........ Where you will write the code for your solution
│   ├── Expr.hs ............ Definition of the Expr type and examples
│   └── Var.hs ............. Definition of the Var helper class (discussed later)
├── test
│   ├── IC
│   │   ├── Approx.hs ...... Testing utilities for approximate equality
│   │   └── Exact.hs ....... Testing utilities for exact equality
│   └── Tests.hs ........... A selection of unit tests written with tasty-hunit
└── calculus.cabal
```

You will work within the `Calculus` module and the tests can be found inside the `Tests` module, as usual. The `Expr` datatype outlined at the start of this specification can be found in the `Expr` module, along with some examples of expressions for you to make use of when testing:

$$
\begin{aligned}
&5 \times x \\
&x \times x + y - 7 \\
&x - y \times y / (4 \times x \times y - y \times y) \\
&-cos(x) \\
&sin(1 + log(2 \times x)) \\
&log(3 \times x \times x + 2)
\end{aligned}
$$

The structure of `test/` has been made more organised to allow tests to check for not only equality (as defined by the `Eq` typeclass), but also approximate equality for dealing with the imprecise nature of

[3] It happens also to be an instance of the *Taylor series* about 0.

[4] See https://gitlab.doc.ic.ac.uk/help/user/ssh.md

floating-point values. Any tests that use `Double` will accept a range of solutions within a small tolerance (notice their test cases use the `(~~>)` operator). The `IC.Exact` and `IC.Approx` contain the functionality to do each of these things. Feel free to expand the test suite with additional cases if you wish. You can test with `cabal test` as usual.

When you use `cabal repl`, you will automatically load the `Calculus` module, and, as this imports `Expr` and `Var`, you will have access to all the exercise's functionality immediately.

What to do

- Define a function `lookUp :: Eq a => a -> [(a, b)] -> b` that will look up the value associated with a key in a list of (key, value) pairs. A precondition is that there is a binding for the key in the list. You should use Haskell's built-in `lookup` function which returns an object of type `Maybe Double`, where

  ```
  data Maybe a = Nothing | Just a
  ```

  The type of `lookup` is:

  ```
  lookup :: Eq a => a -> [(a, b)] -> Maybe b
  ```

  When looking up an item `k`, if the lookup fails the function returns `Nothing`; if it succeeds by finding a binding of the form `(k, v)` in the table it returns `Just v`. In this exercise you may assume that all look-ups succeed, in which case the simplest way to extract the `v` from `Just v` is to use the function `fromJust` from the module `Data.Maybe` which has been imported for you in the template file. The definition looks like this:

  ```
  fromJust :: Maybe a -> a
  fromJust (Just x) = x
  fromJust Nothing = error "..."
  ```

  but you'll never hit the error case.

- Using `lookUp` define a Haskell function `eval :: Expr -> Env -> Double` which, given an expression and an *environment* (`Env`) evaluates the expression, returning a `Double`. The type synonym for `Env` is:

  ```
  type Env = [(String, Double)]
  ```

  We say that `eval` implements an *interpreter* for the expression language defined above.[5]

- This question is optional and will not be auto-tested, but you may find it useful for testing the functions that come later.

  Define a function `showExpr :: Expr -> String` that will generate a neat printable representation for expressions. For example:

  ```
  *Calculus> showExpr (Add (Val 1.0) (Log (Id "x")))
  "(1.0+log(x))"
  *Calculus> showExpr (Mul (Val 4.0) (Add (Id "x") (Neg (Id "y"))))
  "(4.0*(x+-(y)))"
  ```

[5] GHCi is also an interpreter, except that it manipulates the abstract syntax tree of a Haskell program.

Perhaps you can do a bit better than this, e.g. generating "4.0*(x-y)" for the latter. As a challenge try displaying expressions in as simple a form as you can.

As an additional exercise define an explicit instance of the `Show` class for `Expr` so that expressions are always "pretty-printed" by GHCi, e.g.:

```
*Calculus> Add (Val 1.0) (Log (Id "x"))
"(1.0+log(x))"
```

To do this, you will need to comment out the `deriving instance Show Expr` line and define your own instance instead[6].

- Define a function `diff :: Expr -> String -> Expr` that applies the differentiation rules above to a given `Expr`. For example to differentiate $x \times x$, $sin(2x)$ and $1 + log(x)$:

```
*Calculus> diff (Mul (Id "x") (Id "x")) "x"
Add (Mul (Id "x") (Val 1.0)) (Mul (Val 1.0) (Id "x"))

*Calculus> diff (Sin (Mul (Val 2.0) (Id "x"))) "x"
Mul (Cos (Mul (Val 2.0) (Id "x")))
    (Add (Mul (Val 2.0) (Val 1.0)) (Mul (Val 0.0) (Id "x")))

*Calculus> diff (Add (Val 1.0) (Log (Id "x"))) "x"
Add (Val 0.0) (Div (Val 1.0) (Id "x"))
```

which, in a very long-winded way, tells us that

$$\begin{aligned} \frac{d}{dx} x \times x &= x \times 1 \;+\; x \times 1 \quad (\equiv\; 2x) \\ \frac{d}{dx} sin(2x) &= cos(2x) \times (2 \times 1 + 0 \times x) \quad (\equiv 2\,cos(2x)) \\ \frac{d}{dx} 1 + log(x) &= 0 + 1/x \quad (\equiv\; 1/x) \end{aligned}$$

If you've defined `showExp` above you can use it here to make it easier to check your function's output. For example,

```
*Calculus> showExp (diff (Add (Val 1.0) (Log (Id "x"))) "x")
"(0.0+(1.0/x))"
```

Notice how the results returned by `diff` contain a lot of redundancy, e.g. addition of 0.0 and multiplication by 1.0. This is a result of applying the chain rule recursively and without optimisation. Optimisation is the subject of the extension below.

- Define a function `maclaurin :: Expr -> Double -> Int -> Double` that will approximate the value of a function (`Expr`) at a specified point (`Double`) using the first n (`Int`) terms of the Maclaurin series. For example:

[6] This is a "standalone deriving clause", which is not possible in vanilla Haskell, and requires `{-# LANGUAGE StandaloneDeriving #-}` to be present at the top of the file.

```
*Calculus> sin 2
0.9092974268256817
*Calculus> maclaurin (Sin (Id "x")) 2 3
2.0
*Calculus> maclaurin (Sin (Id "x")) 2 5
0.6666666666666667
*Calculus> maclaurin (Sin (Id "x")) 2 7
0.9333333333333333
*Calculus> maclaurin (Sin (Id "x")) 2 9
0.9079365079365079
```

You'll find that things get really slow as *n* gets much bigger because of the redundancies mentioned above. You can fix that later if you have time.

Hints Try using `iterate` together with your `diff` function to generate the infinite list of the higher-order differentials of a given expression. You might also like to look up `scanl` which you can use to generate the list of factorials, `[1, 1, 2, 6, ...]` and maybe also `zipWith`, which you can use to pair-wise combine two lists. There are lots of interesting ways to do this.

- A really neat way to simplify functions like `diff` is to *overload* Haskell's existing functions, like `+`, `*`, `cos` etc. so that they generate `Expr` values, rather than numbers. To do this, make `Expr` an instance of `Num`, `Fractional` and `Floating` and provide overloaded definitions of the `Num` functions `fromInteger`, `negate`, `+` and `*`, the `Fractional` functions `fromRational` and `/`, and the `Floating` functions `sin`, `cos` and `log`.

  Instances for these classes are provided in the template that include definitions for all the member functions, each of the form `fun = undefined`. In practice, there are many more member functions in these classes, but they have been omitted for clarity: this is normally a really bad idea, and GHC will normally warn you, but this warning has been explicitly disabled in the `calculus.cabal` file – in future, implement everything, even if by setting it to `undefined`!

  Now change your implementation of `diff` replacing the constructors on the right-hand side of each rule with the corresponding (overloaded) operator or function. For example, you should find that you can replace `Add e1 e2` with `e1 + e2`. Before doing this you are advised to make a copy of your original version and comment it out using a comment block `{- like so -}`.

Aside As a further piece of "syntactic sugar" the `Vars` module also includes this:

```
class Vars a where
  x, y, z :: a

instance Vars Exp where
  x = Id "x"
  y = Id "y"
  y = Id "z"
```

which means, for example, that `x`, `y` and `z` can be used as shorthand for `Id "x"`, `Id "y"` and `Id "z"` respectively. However, there is a catch: how does GHC know that `x` is an `Expr`, for example? Answer: it doesn't! There could be any number of other `Var` instances, for example:

```
instance Vars Double where
  x = 4.3
  y = 9.2
  z = -1.7
```

in which case `x + cos x :: Double` could either mean either 3.8992008279200245 or `Add (Id "x") (Cos (Id "x"))` depending on which type we pick for `x`.

In the absence of any additional type information we have to help GHC out by telling it which instance to pick, by *forcing* the type using `::`. For example:

```
*Calculus> z :: Double
-1.7
*Calculus> z :: Exp
Id "z"
*Calculus> x + cos x :: Exp
Add (Id "x") (Cos (Id "x"))
*Calculus> showExp (diff (2 * sin (x + log y)) "x")
"((2.0*cos((x+log(y))))+(0.0*sin((x+log(y)))))"
*Calculus> 2 - sin y
1.7771100858997524
```

These examples omit the parentheses around the arguments to `negate`, `sin`, `cos` and `log`. You can put them in if you want: `f x` and `f(x)` mean the same in Haskell.

Notice that we didn't need to force the type when using `showExpr` because `showExpr :: Expr -> String`. This tells the type system that the argument *must* be an `Expr` rather than a `Double`.

What about the last example? How did GHCi know to pick type `y :: Double`? Normally, when the type system can't determine which instance to pick it complains with an "Ambiguous type variable" error message. However, Haskell supports the notion of *default* types and it so happens that the default type for `Fractional` is `Double`. Thus, rather than rejecting expressions like `z` and `2 - sin y` it instead picks the default type `Double`[7].

Optional Extension

You'll notice from some of the examples above that there is a lot of redundancy in the expressions generated by `diff`. To fix this, modify your `Num` and Fractional instances for `Expr` to apply the following

[7] You may already have noticed that integer arithmetic defaults to use type `Integer` rather than, say, `Int`. If you're in doubt try typing `91^131` at the GHCi prompt, for example.

rules *as expressions are being built*:

$$\begin{aligned} -0 &\equiv 0 \\ e+0 = 0+e &\equiv e \\ e*1 = 1*e &\equiv e \\ 0/e &\equiv 0 \\ e/1 &\equiv e \end{aligned}$$

These optimisations will make a big difference to the size of the expressions generated by `diff` and will allow you to compute Maclaurin series much more efficiently. Try this out by running the above `maclaurin` function again, but this time with a much larger number of terms, e.g.

```
*Calculus> sin 2
0.9092974268256817
*Calculus> maclaurin (Sin (Id "x")) 2 9
0.9079365079365079
*Calculus> maclaurin (Sin (Id "x")) 2 15
0.9092974515196738
*Calculus> maclaurin (Sin (Id "x")) 2 30
0.9092974268256817
```

Submission

Once you have `diff` working with overloaded operators/functions you can delete the original version prior to submission.

As with all previous exercises, you will need to use the commands `git add`, `git commit` and `git push` to send your work to the GitLab server. Then, as always, log into the LabTS server, `https://teaching.doc.ic.ac.uk/labts`, click through to your `HaskellCalculus` exercise `https://gitlab.doc.ic.ac.uk/lab2324_autumn/haskellcalculus_username` and request an auto-test of your submission.

IMPORTANT: Make sure that you submit the correct commit to Scientia.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.
```

A: There are no obvious deficiencies in the solution or the student's coding style. In addition, there is evidence of productive testing.

A*: As for an A -- plus the student has done additional work beyond the basic spec, e.g. by considering (and clearly commenting) interesting variations or extensions to the given functions; e.g. based on their own research.