Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Иркутский государственный университет»
(ФГБОУ ВО «ИГУ»)
Институт математики и информационных технологий
Кафедра информационных технологий

ОТЧЕТ
по курсовой работе

ВЫЧИСЛЕНИЕ ПРОИЗВОДНОЙ РАЗЛИЧНЫХ ФУНКЦИЙ И РЕШЕНИЕ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ МЕТОДОМ ЭЙЛЕРА НА ЯЗЫКЕ HASKELL

Студента 3 курса группы 02321-ДБ
направления 01.03.02 «Прикладная
математика и информатика»
Петунина Дениса Евгеньевича

Руководитель:
К. т. н., доцент
Черкашин Евгений Александрович
Оценка ____________

Иркутск – 2022

СОДЕРЖАНИЕ

1 .ВВЕДЕНИЕ

Реализация программ будет выполнена на языке программирования Haskell. Важной особенность языка является поддержка ленивых вычислений, с ними можно делать невозможные вещи. для других языков программирования. Обращаться к ещё не вычисленным значениям, работать с бесконечными данными.

Обычно мы пишем программу, чтобы решить какую-нибудь сложную задачу. Часто так бывает, что сложная задача оказывается сложной до тех пор пока её не удаётся разбить на отдельные независимые подзадачи. Мы решаем задачи по-меньше, потом собираем из них решения, из этих решений собираем другие решения и вот уже готова программа.

Об этом говорит John Huges в статье "Why functional programming matters". Он приводит такую метафору. Если мы делаем стул и у нас нет хорошего клея. Единственное что нам остаётся это вырезать из дерева стул целиком.

Гораздо проще сделать отдельные части и потом собрать вместе.

Функциональные языки программирования предоставляют два новых вида "клея". Это функции высшего порядка и ленивые вычисления.

2 .Дифференцирование

Найдём производную функции в точке. Посмотрим на математическое определение производной:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Производная это предел последовательности таких отношений, при h стремящемся к нулю. Если предел сходится, то производная определена. Для того чтобы решить эту задачу начнём с небольшого значения h и будем постепенно уменьшать его, вычисляя промежуточные значения производной. Как только они перестанут сильно изменяться мы будем считать, что мы нашли предел последовательности

Пример реализации на Haskell:

```haskell
sequenceConvergence :: (Ord q, Num q) => q -> [q] -> q
sequenceConvergence eps (a:b:xs)
    | abs (a - b) <= eps     = a
    | otherwise              = sequenceConvergence eps (b:xs)
```

Теперь сделаем последовательность значений производных.
Напишем функцию, которая вычисляет промежуточные решения:

```haskell
easyDifferentiation :: Fractional q => (q -> q) -> q -> q -> q
easyDifferentiation f x h = (f (x + h) - f x) / h
```

Теперь добавим функцию, которая будет брать изначальное значение шага и уменьшать его вдвое

```haskell
halves :: Double -> [Double]
halves = iterate (/2)
```

Далее соберем все вместе и получим следующее:

```haskell
differentiation :: (Ord q, Fractional q) => q -> q -> (q -> q) -> q -> q
differentiation h0 eps f x = sequenceConvergence eps
```

```
        $ map (easyDifferentiation f x)
        $ iterate (/2) h0
where easyDifferentiation f x h = (f (x + h) - f x) / h
```

Проверим работоспобность нашей функции в интерпритаторе GHCi. Протестируем решение на экспоненте, т.к ее производная равна самой себе:

```
ghci> :l Differentiation.hs
[1 of 1] Compiling Differentiation  ( Differentiation.hs, interpreted )
Ok, one module loaded.
ghci> let exp' = differentiation 1 1e-5 exp
ghci> let test x = abs $ exp x - exp' x
ghci> test 5
1.767240203776055e-5
ghci> test 10
2.550578210502863e-5
```

Рисунок 2.1 – Вывод

2.1 Решение дифференциального уравнения методом Эйлера

Метод Эйлера — простейший численный метод решения систем обыкновенных дифференциальных уравнений.
Метод Эйлера является явным, одношаговым методом первого порядка точности. Он основан на аппроксимации интегральной кривой кусочно-линейной функцией, так называемой ломаной Эйлера.

Погрешность на шаге или локальная погрешность — это разность между численным решением после одного шага вычисления y_i и точным решением в точке $x_i = x_{x-1} + h$.
Численное решение задаётся формулой:

$$y(x_{i-1} + h) = y(x_{i-1}) + hy'(x_{i-1}) + O(h^2)$$

Этой теоритической основы нам хватит чтобы написать реализацию метода Эйлера на языке Haskell.

2.1.1 Реализация метода Эйлера

Напишем функцию, которая принимает на вход саму функцию, шаг и условие выхода:

```
data EulerState = EulerState !Double !Double !Double !Double deriving(Show)
type EulerFunction = Double -> Double

euler :: EulerState -> EulerFunction -> Double -> EulerState
euler (EulerState p v a t) f dstep = (EulerState p' v' a' t')
    where t' = t + dstep
          a' = f t'
          v' = v + a'*dstep
          p' = p + v'*dstep
```

Далее, напишем функцию, которая будет использовать функцию, написанную выше, для решения уравнения, каждый раз приблежая решение и уменьшая погрешность. Условием выхода из рекурсии будет $t < limit$

```
runEuler :: EulerState -> EulerFunction -> Double -> Double -> EulerState
runEuler s f dstep limit = go s
where go s@(EulerState _ _ _ t) = if t < limit then go (euler s f dstep) else s
```

Вот и все. Все что нужно для решения уравнений методом Эйлера сделано.
Осталось только вызвать саму функцию $runEuler$ и передать в нее аргументы.
Функцию будем использовать: $(**2)$
Шаг будем использовать"$1e - 8 = 0,00000001$
и limit = 5
Для этого вызовем функцию в main:

```
main = do
  let limit = 5
      dstep = 1e-8 in print $ runEuler (EulerState 0 0 0 0) (**2) dstep limit
```

И получаем вывод:

```
EulerState 52.08333 41.6666 25.0000 5.0000
```

2.2 Оптимизация реализации метода Эйлера

Реализация метода Эйлера выше, хоть и работает правильно, но работает очень медленно.

real 1m34.753s
user 0m0.000s
sys 0m0.015s

Для выполнения данной программы потребовалось целых 1.5 минуты, что довольно медленно, для такой тривиальной задачи.
Чтобы оптимизировать данную задачу, скомпилируем данный модуль с помощью ключа $-O3(Optimisationflags)$

GHC -03 euler.hs

И получим результат:

real 0m16.173s
user 0m0.000s
sys 0m0.015s

Флаг оптимизации позволил нам сократить время исполнения программы в 6 раз. Проблема настолько медленной программы в функции (**2) - это медленная функция, поскольку (**2) имеет результаты отличные от $x \rightarrow x * x$.
Также функция $runEuler$ является рекурсивной функцией, следовательно, она не может быть встроеной($inlined$). Это означает, что переданная функция также не может быть встроена. Аргументы $dstep$ и $limit$ также передаются для каждой итерации. То, что функция не может быть встроена, означает, что в рекурсии ее аргумент должен быть упакован($boxed$) перед передачей в функцию, а затем его результат должен быть распакован($unboxed$), а это очень дорогая операция.

2.3 Вычисление производных

В пункте 2 Дифференцирование мы уже разобрали вычисление производной через предел отношения приращения функции к приращению её аргумента при стремлении приращения аргумента к нулю. В этой главе мы разберем пример реализации программы, которая преобразует выражение производной функции с помощью таблицы производных элементарных функций. Для начала определим фундаментальные операции для выражений:

```
data MathExpr a
  = X
  | Coef a                                  %Константа, коэфициент
  | Sum (MathExpr a) (MathExpr a)           %Операция суммы
  | Prod (MathExpr a) (MathExpr a)          %Операция умножения
  | Div (MathExpr a) (MathExpr a)           %Операция деления
  | Exp (MathExpr a)                        %Функция экспоненты
  | Log (MathExpr a)                        %Функция логарифма
  deriving (Eq, Show, Read)
```

Далее введем функцию $Value$ которая принимает выражение и вычисляет его.

```
value :: (Floating a, Eq a) => MathExpr a -> a -> a
value (Coef a) n   = a
value X n          = n
value (Sum a b) n  = (value a n) + (value b n)
value (Prod a b) n = (value a n) * (value b n)
value (Div a b) n = (value a n) / (value b n)
value (Exp a) n    = exp $ value a n
value (Log a) n    = log $ value a n
```

Следущее что нам нужно, это парсер выражений, для того, чтобы преобразовать более сложные выражения в простейшие операции. Эта функция берет простейшие случаи каждой возможности выражения и преобразует выражение этих случаев в более простую форму.
Функция может либо возвращать коэффициент, если это возможно, либо просто возвращает упрощенное выражение, которое не является просто числом.

```haskell
parser :: (Floating a, Eq a) => MathExpr a -> MathExpr a
parser (Coef n) = Coef n
parser X = X
parser (Sum (Coef 0.0) v) = parser v
parser (Sum u (Coef 0.0)) = parser u
parser (Sum u v) =
  let uPrime = parser u
      vPrime = parser v
   in if uPrime == u && vPrime == v
        then Sum u v
        else parser $ Sum uPrime vPrime
parser (Prod (Coef 0.0) v) = Coef 0.0
parser (Prod u (Coef 0.0)) = Coef 0.0
parser (Prod (Coef 1.0) v) = parser v
parser (Prod u (Coef 1.0)) = parser u
parser (Prod u v) =
  let uPrime = parser u
      vPrime = parser v
   in if uPrime == u && vPrime == v
        then Prod u v
        else parser $ Prod uPrime vPrime
parser (Div u (Coef 1.0)) = parser u
parser (Div u v) =
  let uPrime = parser u
      vPrime = parser v
   in if uPrime == u && vPrime == v
        then Div u v
        else parser $ Div uPrime vPrime
parser (Exp (Coef 0.0)) = Coef 1.0
parser (Exp u) =
  let uPrime = parser u
   in if uPrime == u
        then Exp u
        else parser $ Exp uPrime
```

```
parser (Log (Coef 1.0)) = Coef 0.0
parser (Log u) =
  let uPrime = parser u
   in if uPrime == u
        then Log u
        else parser $ Log uPrime
```

Далее напишем функцию дифференцирования для нахождения наклона функции в определенной точке. Если функция достаточно сложна, она вернет другую функцию, предназначенную для этого наклона в точке графика, используя таблицу производных элементарных функций.

```
differentiation :: (Floating a, Eq a) => MathExpr a -> MathExpr a
differentiation X = Coef 1.0
differentiation (Coef n) = Coef 0.0
differentiation (Sum a b) = Sum (differentiation a) (differentiation b)
differentiation (Prod a b) = Sum (Prod (differentiation a) (b)) (Prod (a) (differentiatio
differentiation (Div a b) =
  Div
    (Sum (Prod (differentiation a) (b)) (Prod (Coef (-1.0)) (Prod (a) (differentiation b))
    (Prod b b)
differentiation (Exp a) = Prod (Exp a) (differentiation a)
differentiation (Log a) = Div (differentiation a) a
```

В заключение напишем функцию ввода и вывода. Она будет принимать 2 аргумента пути к файлам. Функция найдет первый указанный файл и прочитает его. Если этот файл содержит математическое выражение, то функция прочитает его, дифференцирует, упростит и запишет во выходной файл.

```
readdifferentiationWrite :: FilePath -> FilePath -> IO ()
readdifferentiationWrite fileIN fileOUT =
  if (fileIN == fileOUT)
    then error "The input and output files must be differentiationerent."
    else do
      fileContents <- readFile fileIN
      if fileContents == ""
```

```
then error "Your input file is empty."
else writeFile fileOUT (iodifferentiationLines $ lines fileContents)
```

Теперь проверим нашу программу на функции:

$$(Ln(x^2 - 1))'$$

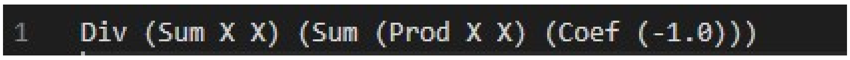
```
Log (Sum (Prod (X) (X)) (Coef (-1.0)))
```

Рисунок 2.2 – Ввод in.txt

Ожидаемый вывод:

$$\frac{2x}{x^2 - 1}$$

Реальный вывод:



```
Div (Sum X X) (Sum (Prod X X) (Coef (-1.0)))
```

Рисунок 2.3 – Вывод out.txt

Что соответствует ожидаемому выводу.

.ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы был представлен малый из возможного функционала языка программирования Haskell. В результате мы получили реализацию методов и программ для решения дифференциальных уравнений методом Эйлера, нахождение производной и преобразование выражений. Целью курсовой работы было показать навыки работы с данным языком, а также методы поиска решений дифференциальных уравнений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Арнольд В. И.Обыкновенные дифференциальные уравнения.: Новое издание, исправл. — М.: МЦНМО с.: ил

2. John Hughes Why Functional Programming Matters: The University, Glasgow

3. Simon Thompson Haskell: the Craft of Functional Programming

4. M. DOUGLAS MCILROY: Power series, power serious Journal of Functional Programming 1999 Cambridge University Press