

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования

**«Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина»**

**Институт математики и компьютерный наук
Кафедра алгебры и дискретной математики**

СИСТЕМА КОМПЬЮТЕРНОЙ АЛГЕБРЫ ДЛЯ .NET FRAMEWORK

«Допущен к защите»

«__» _____ 2013 г.

Квалификационная работа на степень
магистра наук по направлению
010200 Математические основы
компьютерных наук
студента группы МКМ-210119
МЕДВЕДЕВА
ИГОРЯ СЕРГЕЕВИЧА

Научный руководитель –
доцент ОКУЛОВСКИЙ
ЮРИЙ СЕРГЕЕВИЧ,
доцент кафедры
алгебры и дискретной
математики,
кандидат физ.-мат. наук

Екатеринбург
2013

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Описание системы	5
1.1 Структура	5
1.2 Использование	6
2 Определение правил	9
2.1 Общая архитектура правил	9
2.2 Выборка	9
2.3 Фильтрация	12
2.4 Модификация	14
2.5 Задание правила	14
3 Упрощение алгебраических выражений	16
3.1 Описание	16
3.2 Практическое применение	20
4 Дифференцирование в частных производных	21
4.1 Теоретическая часть	21
4.2 Практическая реализация	22
5 Алгоритм численной регрессии	26
5.1 Понятие регрессии	26
5.2 Метод наименьших квадратов	27
5.3 Реализация алгоритма	28
6 Правило резолюций	31
6.1 Построение дерева выражений для логики первого по- рядка	31
6.2 Алгоритм унификации	32
6.3 Задание правила резолюции	34
ЗАКЛЮЧЕНИЕ	36
ЛИТЕРАТУРА	37
ПРИЛОЖЕНИЕ	38

ВВЕДЕНИЕ

Система компьютерной алгебры (СКА) — это приложение для символьных вычислений, то есть выполнения преобразований и работы с математическими выражениями в аналитической (символьной) форме. Типичными примерами таких вычислений являются задачи упрощения алгебраических выражений, дифференцирования в частных и полных производных, интегральные преобразования и многие другие. Такие системы широко используются в математике и компьютерных науках.

Одной из отличительных особенностей СКА является то, что математические структуры представлены в символьном виде. Для работы с данными в такой форме чаще всего используются синтаксические деревья. Пример такого дерева изображен на рис. 1. Узлы x и y обозначают переменные, 5 — это константа, а остальные узлы представляют операции сложения, умножения и функцию натурального логарифма.

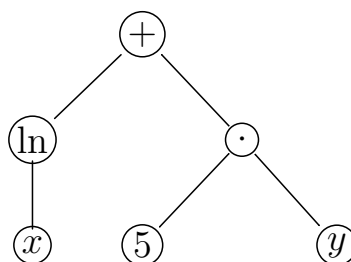


Рис. 1: Синтаксическое дерево

Для реализации символьных вычислений необходимо некоторым образом изменять синтаксические деревья. Один из способов это осуществить — задать некоторое множество правил трансформации и для различных задач использовать подходящие наборы правил. Именно поэтому основной целью этой работы стало создание удобного и надежного инструмента для описания правил трансформации. Кроме того, необходимо было проверить практическую ценность данного подхода и протестировать его на актуальных задачах СКА. Такими задачами стали:

- Задача упрощения алгебраических выражений

- Задача дифференцирования в частных производных
- Реализация алгоритма численной регрессии с использованием символьных вычислений
- Построение синтаксического дерева для логических выражений в Сколемовской нормальной форме и задание для них правила резолюции

Решение разрабатывалось для платформы *.NETFramework* [1] на языке *C#*. В настоящее время есть несколько похожих систем для этой платформы, однако, по имеющимся данным, не одна из них не предоставляет функционала, реализованного в рамках данной работы.

1 Описание системы

1.1 Структура

Систему можно условно разделить на следующие основные логические части:

- Синтаксическое дерево
 - Узлы дерева (INode)
 - * `ConstantNode`: узлы с константным значением
 - * `VariableNode`: переменные
 - * `ITermNode`: термы в логике первого порядка
 - `FunctionNode`: функции (нульарные функции являются константами)
 - `PredicateNode`: предикаты
 - Операции
 - * Унарные
 - Арифметическое отрицание
 - Синус
 - Косинус
 - Тангенс
 - Натуральный логарифм
 - Логическое отрицание
 - * Бинарные
 - Сложение
 - Вычитание
 - Произведение
 - Отношение
 - Возведение в степень
 - Дифференцирование
 - Логическое И

- Логическое ИЛИ
- * Тернарные
- * Мультиарные
 - Множественная дизъюнкция. Используется как сколемовская нормальная форма в правиле резолюции
 - Множественная конъюнкция
- Задание правил
- Коллекции правил
- Численная регрессия
- Правило резолюции
 - Сервис унификации
- Вспомогательные классы
 - Сервис для работы с коллекциями правил
 - Парсеры
 - * `Tree2Expression`: синтаксическое дерево \rightarrow `Expression`
 - * `Expression2Tree`: `Expression` \rightarrow синтаксическое дерево
 - * `Expression2LogicTree`: `Expression` \rightarrow логическое дерево
- Внешние сервисы
 - Сервис, содержащий все основные функции системы.
 - Генератор случайных алгебраических выражений (в виде синтаксических деревьев)
- Тесты

1.2 Использование

Данная система изначально разрабатывалась для использования в сторонних проектах в качестве библиотеки символьных вычислений. Этим объясняется наличие различных сервисов, упрощающих использование основных

функциональностей системы, и нескольких реализаций одной функциональности с разной сигнатурой. Как именно использовать конкретную функциональность будет описано в соответствующих разделах. Здесь же представлена наиболее общая информация, которая будет полезна любому пользователю, независимо от того, какие цели он преследует, используя систему.

Первое, что нужно сделать — это добавить в проект библиотеку `ComputerAlgebra.dll` и определенным образом задать входные данные (алгебраическое выражение или логическую формулу), которые в дальнейшем должны быть преобразованы в синтаксические деревья. Конечно, выражения можно задавать в строковом виде, однако такой подход чреват возможными синтаксическими ошибками, которые проявятся только во время выполнения программы. Поэтому было решено использовать стандартное средство платформы .NET для работы с синтаксическими деревьями - **Expression Trees**. Это позволяет задавать алгебраические выражения в удобном и наглядном виде. Например функция $f(x, y) = (x - 5)^2 + y^2 - 25$ задается следующим лямбда-выражением:

$$(x, y) => \text{Math.Pow}((x - 5), 2) + \text{Math.Pow}(y, 2) - 25$$

Подробнее про то, как таким же образом задавать формулы логики предикатов, описано в разделе 6.

Задав входные данные в виде **Expression Trees**, их нужно преобразовать во внутренний формат деревьев системы **INode** с помощью соответствующих парсеров (см. 1). Однако для использования некоторых функций системы, таких как упрощение или дифференцирование, это необязательно.

Дальнейшее использование системы тривиально: в большинстве случаев достаточно вызвать один статический метод. Следующий листинг демонстрирует простоту нахождения частной производной (более подробно в разделе 4):

```

using System;
using System.Linq.Expressions;
using AIRLab.CA;
using AIRLab.CA.Tools;
namespace DifferentiatedDemo
{
    class DifferentiatedDemo
    {
        protected delegate double del3(double p1, double p2,
            double p3);
        static void Main()
        {
            Expression<del3> function = (x, y, z) =>
                Math.Pow(x, 3)*y - Math.Pow(z, x)/(x+y)+5*z;
            var result = ComputerAlgebra.Differentiate
                (function, variable : "x");
            Console.WriteLine(result);
        }
    }
}

```

Листинг 1.1: Программа, вычисляющая частную производную функции

$$f(x, y, z) = yx^3 - \frac{z^x}{x+y} + 5z \text{ по переменной } x$$

2 Определение правил

2.1 Общая архитектура правил

Применение правила можно разделить на несколько стадий. Первой стадией является *выборка*. В стадии выборки система принимает одно или несколько выражений, которые представлены в виде синтаксических деревьев, и генерирует кортежи операций, связанные отношениями вложенности. В стадии *фильтрации* происходит фильтрация кортежей по некоторым критериям. В заключительной стадии *модификации* система преобразует дерево в соответствии с правилом. В самом распространенном случае правило применяется для одного дерева. Для таких *унарных* правил, начальное дерево копируется, а затем его копия преобразуется в соответствии с правилом. Однако в некоторых случаях правило может обрабатывать несколько деревьев. Например, в логическом выводе **modus ponens** правило принимает два дерева $A \rightarrow B$ и A и производит B . В такой ситуации новое дерево создается из копий узлов выбранного кортежа. Благодаря тому, что система всегда копирует узлы до преобразований, корректность начальных деревьев сохраняется.

2.2 Выборка

Чтобы выполнить выборку, необходимо определить, какую структуру мы ищем в дереве. Кроме того, нужно уметь формировать кортежи из узлов этой структуры. Для этого используются строки запроса специально разработанного вида. Они являются выражениями, порожденными следующей грамматикой (литералы A, B, C , $.$ и $?$ являются терминалами):

$$S \Rightarrow ML \mid ML(S) \mid S, S$$

$$M \Rightarrow ? \mid . \mid \lambda$$

$$L \Rightarrow A \mid B \mid C$$

Рассмотрим синтаксис строк запроса и их семантику на примере дерева, изображенного на рисунке 2.1.

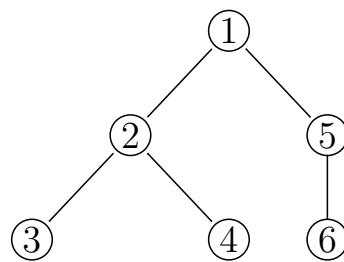


Рис. 2.1: Синтаксическое дерево

В таблице 2.1 представлены строки запроса с соответствующими кортежами и комментариями.

Запрос	Кортежи	Описание
A	(1)	Корень дерева
?A	(1),(2),(3),(4),(5),(6)	Произвольный узел дерева
?A(B)	(5,6)	Произвольный узел в дереве, который имеет единственного потомка, и этот потомок
?A(B,C)	(1,2,5), (2,3,4)	Произвольный узел в дереве, который имеет ровно двух потомков, и эти потомки в заданном порядке
?A(.B,.C)	(1,2,5), (1,5,2), (2,3,4), (2,4,3)	Произвольный узел, который имеет не менее двух потомков, и всевозможные их перестановки
?A(.B)	(1,2), (2,3), (2,4), (5,6)	Произвольный узел с минимум одним потомком, и все эти потомки
?A(?B)	(1,2), (1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (5,6)	Произвольный узел и всевозможные его потомки
?A(?B(?C))	(1,2,3), (1,2,4), (1,5,6)	Аналогичные кортежи для трех степеней вложенности

$?A(?B(C,D))$	$(1,2,3,4)$	Произвольный узел, произвольный потомок которого имеет ровно два потомка
$?A(?B(.C,.D))$	$(1,2,3,4), (1,2,4,3)$	Произвольный узел, произвольный потомок которых имеет ровно двух потомков, и эти потомки следуют в произвольном порядке
$A, ?B$...	Производит выборку по двум деревьям, применяя к первому дереву запрос A , а ко второму - запрос $?B$, и объединяя результаты по принципу FULL JOIN

Таблица 2.1: Примеры строк запроса

Реализация функциональности строк запроса осуществляется классами **BasicSelector** и **ComplexSelector**. Первый класс обрабатывает одно дерево строкой поиска, а второй несколько деревьев, каждое из которых обрабатывается с помощью **BasicSelector**, после чего осуществляется объединение кортежей. Алгоритм работы **BasicSelector** сводится к обходу дерева в глубину. Обработывая произвольный узел, алгоритм работает по следующим правилам:

- если строка запроса имеет вид (A_1, \dots, A_n) , то алгоритм проверяет, что у узла ровно n потомков, и присваивает A_i соответствующего потомка.
- если строка запроса имеет вид $(.A_1, \dots, .A_n)$, то алгоритм проверяет, что у узла n или больше потомков. Затем находятся все возможные комбинации A_i и потомков узла, и происходит соответствующее присвоение. Для всех таких присвоений запускается дальнейший поиск.
- если строка запроса имеет вид $(?A_1, \dots, ?A_n)$, алгоритм присваивает A_i все возможные комбинации потомков узла. Для всех таким присвоений запускается дальнейший поиск.

Для задания правил необходимо каким-то образом определить строку запроса в коде программы. Конечно, это можно сделать в виде строки, но это неудобно по нескольким причинам:

1. Изначально было понятно, что правил потребуется много, и, следовательно, разработка каждого из них не должна занимать много времени.
2. При кодировании правил неизбежны ошибки, поэтому необходимо, чтобы код, задающий правила, был легким для восприятия.
3. Для раннего обнаружения ошибок: необходимо разработать систему таким образом, чтобы как можно большее количество ошибок было обнаружено во время компиляции (в compile-time), а не во время исполнения программы (в runtime).

Было разработано решение, позволяющее задавать строки запроса на языке **C#** и контролировать их корректность на этапе компиляции программы. По структуре строка поиска также является деревом. Каждому элементу этого дерева соответствует объект класса **SelectClauseNode**, который описывает букву, соответствующую данному узлу, модификатор этой буквы ('?', '.', или отсутствие модификатора), а также потомков этого элемента. В классе **SelectClauseWriter** определены статические поля **X**, **AnyX** и **ChildX** для различных **X=A,B,C,...** Каждый из этих полей является экземпляром **SelectClauseNode** с соответствующим образом определенной буквой и модификатором. При использовании этих полей в виде **SelectClauseWriter.A**, они являются листовыми элементами строки поиска. При их индексации квадратными скобками **SelectClauseWriter.A[SelectClauseWriter.B]** создается новый экземпляр **SelectClauseNode** с указанным списком потомков. Предполагается, что классы, в которых определяются правила, наследуются от **SelectClauseWriter**, что позволяет задавать запросы в виде **Select(AnyA[ChildB, ChildC])**.

2.3 Фильтрация

Стадию фильтрации можно разделить еще на две под-стадии: фильтрацию по типам и фильтрацию по значениям. В ходе фильтрации по типам

отбрасываются кортежи, элементы которых не соответствуют типам, предусмотренным в правиле. А в ходе фильтрации по значениям дополнительно отбрасываются кортежи, в которых элементы (уже типизированные) не удовлетворяют дополнительным свойствам. Очевидно, что фильтрация по типам должна происходить до фильтрации по значениям, потому что значения узла зависят от его типа. Например, чтобы проверить, что значение константы равняется нулю, необходимо быть уверенным в том, что проверяемый узел дерева действительно является константой. Иерархию узлов можно посмотреть в разделе Описание системы.

Во время разработки возникла проблема, каким образом задавать кортеж для условий фильтрации. Невозможно хранить выбранные узлы в массиве или списке, так как эти структуры не позволяют определять разные типы для элементов. Единственным возможным решением для этих структур является хранение общего типа и последующее приведение элементов к конкретным типам (листинг 2.1).

```
( array => array[0] is Plus && array[1] is Constant
&& (array[1] as Constant).Value==0 )
```

Листинг 2.1: Кортеж в виде массива

Такие запросы существенно менее читаемы, и, кроме того, содержат потенциальные run-time ошибки в касте `array[1] as Constant`: в случае, если разработчик правила забыл проверить, что `array[1] is Constant`, программа выдаст ошибку во время применения правила, которое наступает в непредсказуемый момент. Благодаря использованию таких технологий, как *generic-methods* и *кодогенерация*, пользователь не может совершить эту ошибку: все приведения типов осуществляются на этапе компиляции программы.

Для фильтрации в классе `Rule` предусмотрен метод `Rule.SelectWhere`, который принимает массив `INode[] roots`, выполняет выборку, а затем передает полученное перечисление `IEnumerable<SelectOutput>` делегату `where`, который является делегатом типа `Func<SelectOutput, WhereOutput>`. Данный делегат проверяет соответствие кортежа `SelectOutput` условиям фильтра и генерирует класс `WhereOutput`, соответствующий прошедшему фильтрацию кортежу. Соответственно, результатом работы `Rule.SelectWhere` является перечисление `IEnumerable<WhereOutput>`.

2.4 Модификация

После прохождения фильтрации, кортежи могут быть модифицированы. Здесь необходимо обратить внимание на два важных момента:

1. Лишь один кортеж из перечисления `IEnumerable<WhereOutput>` может быть модифицирован, поскольку при модификации одного кортежа возможна ситуация, когда остальные кортежи перестают удовлетворять условиям выборки или фильтрации.
2. Модификация должна происходить на "чистой копии" обрабатываемого выражения. Желательное поведение системы — при применении правила к выражению A , выражение A не менялось, а вместо этого создавалось выражение B , которое представляет собой копию A , подвергшуюся изменению. Поэтому, перед применением модификации, все входящие в него выражения заменяются копией.

Для обеспечения этой функциональности класс `Rule` содержит метод `Apply`, принимающий аргумент `WhereOutput`, который соответствует кортежу, подвергающемуся изменениям. Далее осуществляется клонирование этого кортежа методом `WhereOutput.MakeSafe()`, который возвращает объект класса `ModInput`, представляющий собой декоратор над безопасной копией кортежа `WhereOutput`. После этого вызывается делегат `Rule.apply` сигнатуры `Action<ModInput>`, который и осуществляет модификацию переданного объекта `ModInput`.

2.5 Задание правила

Полностью задание правила выглядит следующим образом:

```
Rule.New("+0", StdTags.SafeResection, StdTags.Algebraic,
        StdTags.Simplification)
    .Select(AnyA[B, C])
    .Where<Arithmetic.Plus<double>, Constant<double>,
        INode>(z => z.B.Value == 0)
    .Mod(z => z.A.Replace(z.C.Node));
```

Листинг 2.2: Пример правила

Данная инструкция создает класс правил поэтапно. Каждый из методов `New`, `Select`, `Where` и `Mod` возвращают экземпляры классов `NewRule`, `SelectRule`, `SelectWhereRule` и, наконец, `Rule`, которые накапливают введенную в методы информацию.

Метод `New` принимает имя правила, а также набор тэгов, которые описывают место правила в иерархии. Он возвращает объект класса `NewRule`. Этот класс содержит метод `Select`, принимающий в качестве аргументов `SelectClauseNode`, и отвечающий за стадию выборки. В результате работы метод `Select` возвращает объект `SelectRule`, у которого есть генерик-методы `Where`, сгенерированные для разного количества генерик-параметров `T0`, `T1`, и т.д. Они отвечают за фильтрацию кортежей. За заключительную стадию фильтрации отвечает класс `SelectWhereRule` и его метод `Mod`.

3 Упрощение алгебраических выражений

3.1 Описание

Существует большое количество задач, в которых синтаксические деревья выражений строятся случайным образом или по каким-то правилам, гарантирующим только общую корректность получившегося дерева. В качестве примера рассмотрим работу генетического алгоритма в задаче символьной регрессии.

Основная идея подхода генетического программирования — изменение выражений с помощью случайных операций мутаций и скрещивания и отбор выражений, которые наилучшим образом удовлетворяют регрессируемым данным (например, в смысле среднеквадратичного отклонения) до тех пор, пока не будет получено выражение, приемлемым образом описывающее регрессируемые данные. Отличие эволюционных символьных вычислений от других вариантов генетического программирования заключается в том, что мутации и скрещивания проводятся на основании правил, рассмотренных в главе 2. Основная проблема, решаемая предложенным подходом, это ”раздувание” выражений в генетическом программировании — в ходе подбора выражения в нем образуются неэффективные части. Например, в результате операции скрещивания двух деревьев с рис. 3.1, может получиться дерево, изображенное на рис. 3.2, некоторые узлы которого не влияют на значение, кодируемое выражением, но удлиняют его, затрудняя эволюцию.

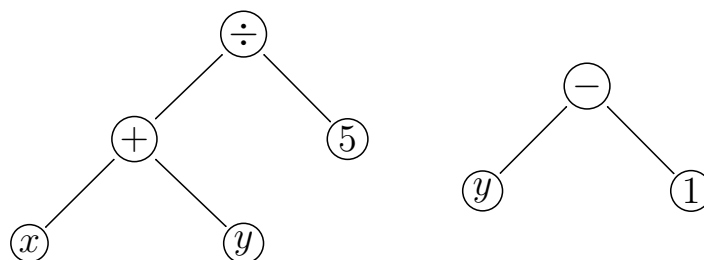


Рис. 3.1: Скрещиваемые деревья

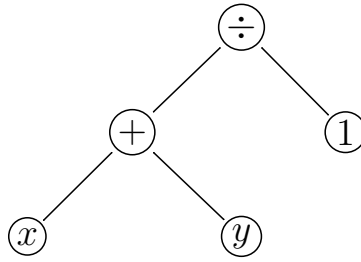


Рис. 3.2: Результат скрещивания

В эволюционных символьных вычислениях используются правила типа $x \div 1 \Rightarrow x$, которые выбрасывают неэффективные участки из выражения. Таким образом, применив правила упрощения к результату предыдущего скрещивания, мы получим выражение, с синтаксическим деревом, изображенным на рис. 3.3.

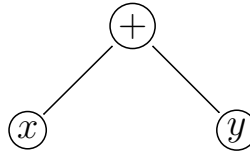


Рис. 3.3: Результат упрощения

Работа алгоритма упрощения заключается в последовательном применении определенных правил к исходному синтаксическому дереву. В сущности, интерес представляет лишь набор правил упрощения, где каждое правило задается уже описанным способом:

```
Rule.New("/1", StdTags.SafeResection, StdTags.Algebraic,
        StdTags.Simplification)
    .Select(AnyA[B, C])
    .Where<Arithmetic.Divide<double>,
        INode, Constant<double>>(z => z.C.Value == 1)
    .Mod(z => z.A.Replace(z.B.Node));
```

В приведенном примере создается правило $/1$, которое является алгебраическим (`StdTags.Algebraic`) правилом упрощения (`StdTags.Simplification`) с безопасным усечением синтаксического дерева (`StdTags.SafeResection`). Схематично оно выглядит так:

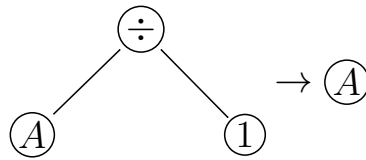


Рис. 3.4: Правило /1

где A - это объект типа `INode`, т.е. любое поддереву, которое и будет результатом применения правила.

Полный список реализованных правил упрощения приведен в таблице 3.1.

Обозначение	Описание
*0	Заменяет нулевой константой все поддеревья с умножением, у которых любой из потомков ноль
*1	Заменяет узел умножения на одного из его потомков, если его второй потомок единица
+0	Заменяет узел сложения на одного из его потомков, если его второй потомок ноль
-0	Заменяет узел разности на его левого потомка, если его правый потомок ноль
0-	Заменяет узел разности на арифметическое отрицание правого потомка, если его левый потомок ноль
/1	Заменяет узел деления на его левого потомка, если его правый потомок единица
0/	Заменяет нулевой константой все поддеревья с делением, у которых левый потомок ноль
0^	Заменяет нулевой константой все поддеревья с возведением в степень, у которых левый потомок ноль
^1	Заменяет узел возведения в степень на его левого потомка, если его правый потомок единица
^0	Заменяет узел возведения в степень на единичную константу, если его правый потомок ноль
C+C	Заменяет узел сложения двух констант на сумму значений этих констант

$C * C$	Заменяет узел произведения двух констант на его результат
$C - C$	Заменяет узел разности двух констант на разность значений этих констант
C / C	Заменяет узел отношения двух констант на его результат
$C ^ C$	Заменяет узел возведения константы в константную степень на результат этой операции

Таблица 3.1: Правила упрощения

Разумеется, этот список правил далеко не полон — существует большое число потенциально упрощаемых выражений, которые не получится упростить, используя только эти правила. Например, если реализовать правило разложения полного квадрата, то можно провести следующую операцию упрощения:

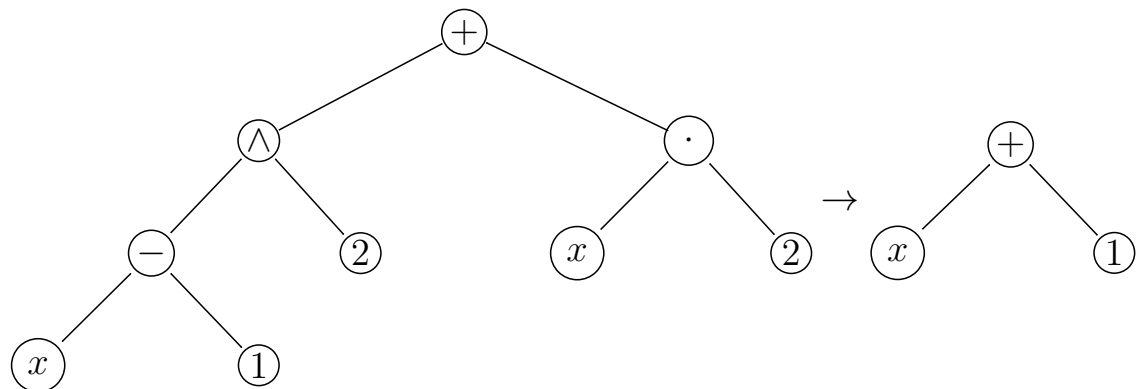


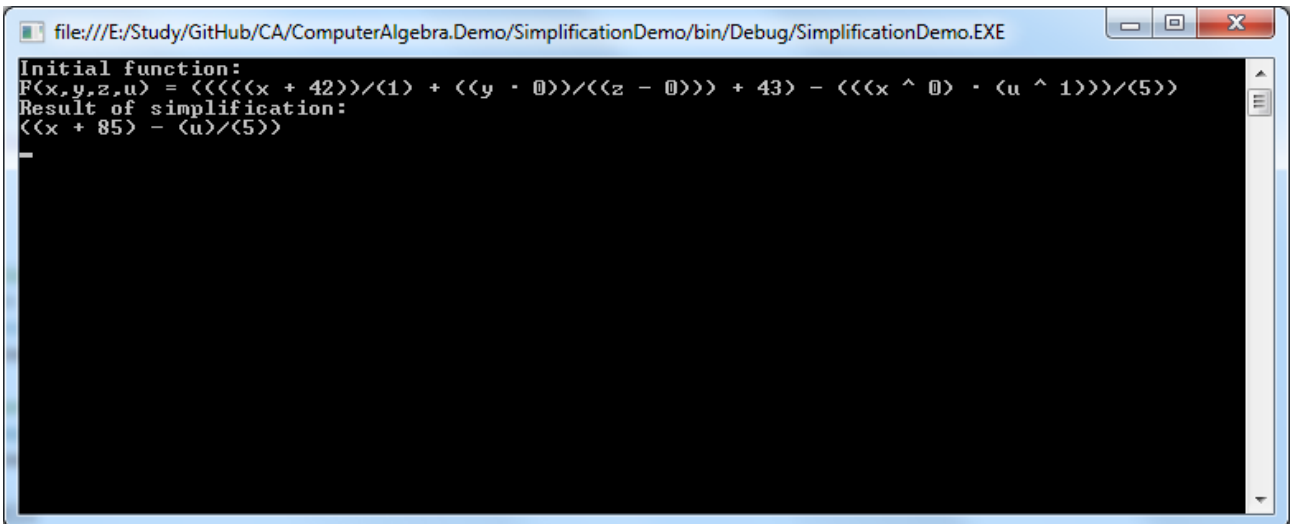
Рис. 3.5: Упрощение полного квадрата

Благодаря тому, что задание правил не представляет никакого труда, система очень хорошо масштабируется. Ее можно использовать как в уже существующем варианте в качестве библиотеки, так и расширив функционал новыми правилами. В следующем подразделе будет рассмотрено, как пользоваться системой на практике.

3.2 Практическое применение

Как уже было сказано ранее, функция упрощения используется в символьных эволюционных вычислениях для удаления из синтаксических деревьев рудиментарных частей, не влияющих на значение, кодируемого выражением. Другое применение — это упрощение дифференциала в методе наименьших квадратов, который будет подробно рассмотрен в главе 5.

Очевидно, что область применения метода достаточно обширна и не ограничивается двумя случаями. Чтобы использовать данную функциональность в своем проекте, необходимо вызвать статический метод `ComputerAlgebra.Simplify`. Этот метод принимает на вход корень синтаксического дерева типа `INode` и возвращает корень упрощенного дерева такого же типа. Кроме того существует дополнительная реализация этого метода, которая принимает и возвращает значения типа `Expression`.



```
file:///E:/Study/GitHub/CA/ComputerAlgebra.Demo/SimplificationDemo/bin/Debug/SimplificationDemo.EXE
Initial function:
F(x,y,z,u) = <<<<<x + 42>>/<1> + <<y - 0>>/<<z - 0>>> + 43> - <<<x ^ 0> - <u ^ 1>>>/<5>>
Result of simplification:
<<x + 85> - <u>/<5>>
```

Рис. 3.6: Пример упрощения функции

4 Дифференцирование в частных производных

4.1 Теоретическая часть

Из математического анализа хорошо известно определение производной функции $y = f(x)$ по переменной x в точке x_0 , как предела отношения приращения функции $\Delta y = f(x_0 + \Delta x) - f(x_0)$ к соответствующему приращению аргумента Δx при $\Delta x \rightarrow 0$, т. е.

$$f'(x) = \lim_{\Delta x \rightarrow 0} \left(\frac{\Delta f(x)}{\Delta x} \right) = \lim_{\Delta x \rightarrow 0} \left(\frac{f(x + \Delta x) - f(x)}{\Delta x} \right) \quad (4.1)$$

Производные играют важнейшую роль в исследовании поведения функций и при нахождении их экстремумов, т. е. в теории и практике оптимизации. Весьма широко используются производные и в интервальных методах, в особенности при решении нелинейных задач. В связи с этим большое значение приобретают способы нахождения производных от функций, и соответствующие алгоритмы часто используются в различных оптимизационных процедурах [2].

В современной вычислительной практике широко применяют три подхода к нахождению производных:

- символьное,
- численное,
- и автоматическое

дифференцирование.

Символьными называют такие вычисления, результаты которых представляются в аналитическом виде. Вычисления в символьном виде отличаются большей общностью и позволяют судить о математических, физических и иных закономерностях решаемых задач.

Именно задача машинного нахождения производной в символьном виде в свое время положила начало исследованиям в области символьных вычислений.

Фактически, задача символьного дифференцирования сводится к синтаксическому разбору выражения с последовательным применением определенных правил. Кроме того, при дифференцировании эти правила достаточно строги и их немного. Например программа на языке Лисп (в свое время задача машинного нахождения производной в символьном виде являлась одной из мотиваций создания нового языка Лисп как языка обработки символов [4]), способная найти производную любой элементарной формулы, занимает всего 40 строк (листинг 1 в Приложении).

Однако результаты ее работы выглядят довольно неожиданно. Попробуем простой пример: найдем производную функции x^2 относительно x :

```
(diff '(expt x, 2) 'x)
(+ (* 1 2 (expt x (- 2 1))) (* 0 (expt x) (log x)))
```

Вероятно, потребуется некоторое время, чтобы понять, что это выражение действительно эквивалентно $2 \cdot x$. Программа не знает, что $2 \cdot 1 = 2$, $2 - 1 = 1$, $0 \cdot x = 0$ и тому подобное. То есть, она не может упростить выражение. В действительности, упрощение одна из главных проблем в компьютерной алгебре [3]. Однако в данной системе функция упрощения уже реализована и для дифференцирования достаточно дополнить набор правил упрощения, правилами взятия производной.

4.2 Практическая реализация

Таким образом, задача дифференцирования свелась к созданию коллекции правил взятия производной известных функций. Для единообразия было решено реализовать функцию дифференцирования в виде бинарной операции. Первым операндом такой операции будет корень дерева алгебраического выражения, а вторым — переменная, по которой будет браться производная. Это позволяет задавать правила дифференцирования в таком же виде, что и правила упрощения:

Rule

```
.New("d(-U)/dx", StdTags.Differentiation, StdTags.Algebraic)
.Select(AnyA[ChildB[ChildC], ChildD])
.Where<Differentiation.Dif<double>, Arithmetic.Negate<double>,
  INode, VariableNode>()
.Mod(z => z.A.Replace(new Arithmetic.Negate<double>(
  new Differentiation.Dif<double>(z.C.Node, z.D.Node)))));
```

Листинг 4.1: Правило дифференцирования $\frac{d(-U)}{dx} = -\frac{dU}{dx}$

Полный список реализованных правил представлен в таблице 4.1.

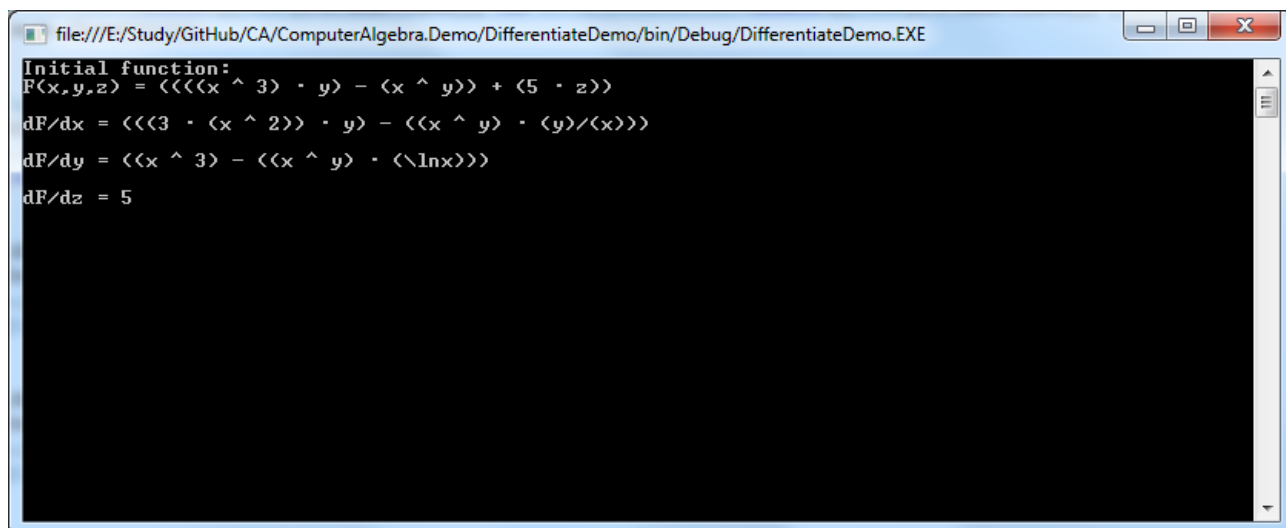
Обозначение	Описание
$d(-U)/dx$	Производная отрицания $\frac{\partial(-U)}{\partial x} = -\frac{\partial U}{\partial x}$
$d(U+V)/dx$	Производная суммы $\frac{\partial(u(x) + v(x))}{\partial x} = \frac{\partial u(x)}{\partial x} + \frac{\partial v(x)}{\partial x}$
$d(U-V)/dx$	Производная разности $\frac{\partial(u(x) - v(x))}{\partial x} = \frac{\partial u(x)}{\partial x} - \frac{\partial v(x)}{\partial x}$
$d(U*V)/dx$	Производная произведения $\frac{\partial(u(x) \cdot v(x))}{\partial x} = \frac{\partial u(x)}{\partial x} \cdot v(x) + \frac{\partial v(x)}{\partial x} \cdot u(x)$
$d(U/V)/dx$	Производная частного $\frac{\partial(u(x) \cdot v(x))}{\partial x} = \frac{\frac{\partial u(x)}{\partial x} \cdot v(x) - \frac{\partial v(x)}{\partial x} \cdot u(x)}{v^2(x)}$
$d(U^c)/dx$	Производная степенной функции $\frac{\partial x^n}{\partial x} = nx^{(n-1)}$, где n - постоянное число
$d(\ln U)/dx$	Производная натурального логарифма $\frac{\partial \ln x}{\partial x} = \frac{1}{x}$

$d(U^V)/dx$	Производная сложной функции вида f^g : $\frac{\partial U^V}{\partial x} = U^V \cdot \left(\frac{\partial V}{\partial x} \cdot \ln U + \frac{V \cdot \frac{\partial U}{\partial x}}{U} \right)$
dx/dx	$\frac{\partial x}{\partial x} = 1$
dy/dx	$\frac{\partial y}{\partial x} = 0$
dc/dx	Производная константы $\frac{\partial C}{\partial x} = 0$

Таблица 4.1: Правила дифференцирования

Основной причиной начала исследований в области символьного дифференцирования стала необходимость реализации алгоритма градиентного спуска, который используется в методе наименьших квадратов для решения задачи численной регрессии. Для этой цели описанной коллекции правил вполне достаточно, однако, как и в задаче упрощения, эта коллекция легко расширяется в случае необходимости.

Процедура использования функциональности дифференцирования практически не отличается от процедуры, описанной в разделе 1.2: необходимо подключить библиотеку `ComputerAlgebra.dll` и вызвать статический метод `ComputerAlgebra.Differentiate`. Результаты работы можно видеть на рисунке 4.1.



```
file:///E:/Study/GitHub/CA/ComputerAlgebra.Demo/DifferentiateDemo/bin/Debug/DifferentiateDemo.EXE
Initial function:
F(x,y,z) = <<<(x ^ 3) * y - (x ^ y) + (5 * z)>>
dF/dx = <<<3 * (x ^ 2) * y - (x ^ y) * (y)/(x)>>
dF/dy = <<<(x ^ 3) - (x ^ y) * (\ln x)>>
dF/dz = 5
```

Рис. 4.1: Пример взятия частной производной

5 Алгоритм численной регрессии

5.1 Понятие регрессии

Во многих экспериментальных работах мы хотим исследовать, как изменения одной переменной влияют на другую. Иногда две переменные связаны точным уравнением прямой линии. Например, если сопротивление R простой цепи поддерживается постоянным, то протекающий ток I меняется линейно при линейном изменении напряжения V в соответствии с законом Ома $I = V/R$. Если бы мы не знали закона Ома, то могли бы найти зависимость эмпирически, изменяя V и измеряя I , поддерживая тем временем R фиксированным. Тогда мы бы увидели, что график зависимости I от V дает более или менее прямую линию, проходящую через начало координат. Если установить связь между зависимой случайной величиной Y и величиной X , то уравнение Y относительно X будет называться *уравнением регрессии* [5].

Однако в природе линейная зависимость встречается крайне редко. Более того, зачастую мы даже не можем точно опередить вид функции регрессии — это может быть как многочлен 3 степени, так и тригонометрическая функция. На практике уравнение регрессии чаще всего ищется в виде линейной функции $Y = b_0 + b_1X_1 + b_2X_2 + \dots + b_NX_N$, наилучшим образом приближающей искомую кривую. Делается это с помощью метода наименьших квадратов, когда минимизируется сумма квадратов отклонений реально наблюдаемых Y от их оценок \hat{Y} (имеются в виду оценки с помощью прямой линии, претендующей на то, чтобы представлять искомую регрессионную зависимость):

$$\sum_{k=1}^{|M|} (Y_k - \hat{Y}_k)^2 \rightarrow \min \quad (5.1)$$

($|M|$ — объём выборки экспериментальных данных). Этот подход основан на том известном факте, что фигурирующая в приведённом выражении сумма принимает минимальное значение именно для того случая, когда $Y = y(x_1, x_2, \dots, x_N)$ [6]

5.2 Метод наименьших квадратов

Задача, решаемая в рамках данной работы, будет отличаться от канонического определения задачи нахождения регрессирующей функции. Предполагается, что функция регрессии уже известна, но константные параметры этой функции подобраны неоптимально.

Пусть функция регрессии $f(x_1 \dots x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n$ уже найдена, т.е. известен набор (c_1, \dots, c_n) . Следует заметить, что функция f здесь указана линейная, однако все следующие рассуждения справедливы и для нелинейной функции. Линейная функция выбрана только для удобства записи формул в дальнейшем. Определим новую функцию G следующим образом:

$$G(x_1, \dots, x_n, a_1, \dots, a_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n \quad (5.2)$$

То есть функция G строится путем замены констант функции f новыми переменными. Еще одно важное уточнение заключается в том, что константами считаются все числовые листы синтаксического дерева, т.е. степень 2 в функции $f(x) = x^2$ так же является константным узлом и при построении G будет заменена на соответствующую переменную.

Набор экспериментальных данных определим как:

$$M = \left\{ (X, Z) \mid X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} Z = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} \right\} \quad (5.3)$$

Таким образом, задача метода наименьших квадратов сводится к минимизации следующей функции:

$$H(a_1, a_2, \dots, a_n) = \sum_{i=1}^{|M|} (G(x_{i1}, \dots, x_{in}, a_1, \dots, a_n) - z_i)^2 \quad (5.4)$$

Решение можно искать различными методами нелинейного программирования, однако в данной работе реализован только градиентный спуск. Этот метод не является оптимальным или быстрым, но он прост в реализации и используется как «*proof of concept*», то есть для проверки корректности данного подхода.

Как и в любом другом градиентном методе, основной идеей является движение вдоль антиградиента. Напомним, что градиентом функции $\varphi(x_1, x_2, \dots, x_n)$ называется n -мерный вектор:

$$\nabla\varphi = \left(\frac{\partial\varphi}{\partial x_1}, \dots, \frac{\partial\varphi}{\partial x_n} \right) \quad (5.5)$$

Таким образом, шаг алгоритма выглядит следующим образом:

$$\vec{a}^{[j+1]} = \vec{a}^{[j]} - \lambda^{[j]} \nabla H(\vec{a}^{[j]}) \quad (5.6)$$

где $\lambda^{[j]}$ выбирается в зависимости от угла α_j между последовательными векторами переходов в процессе спуска от точки A_{j-1} до A_j :

$$\cos \alpha_j = \frac{\sum_{i=1}^n \left(\frac{\partial H}{\partial a_i} \right)_{A_j} \cdot \left(\frac{\partial H}{\partial a_i} \right)_{A_{j-1}}}{\sqrt{\sum_{i=1}^n \left(\frac{\partial H}{\partial a_i} \right)_{A_j}^2} \cdot \sqrt{\sum_{i=1}^n \left(\frac{\partial H}{\partial a_i} \right)_{A_{j-1}}^2}} \quad (5.7)$$

Тогда

$$\lambda_{j+1} = \begin{cases} 2\lambda_j, & \alpha_j < \alpha_{min} \\ \lambda_j, & \alpha_{min} \leq \alpha_j \leq \alpha_{max} \\ \frac{1}{3}\lambda_j, & \alpha_j > \alpha_{max} \end{cases} \quad (5.8)$$

В качестве α_{min} и α_{max} взяты углы в 30° и 90° соответственно [7].

5.3 Реализация алгоритма

Алгоритм реализован в классе `RegressionAlgorithm`. Конструктор класса принимает на вход 5 параметров:

- `INode expression` - синтаксическое дерево функции регрессии

- `List<double[]> arguments` - аргументы, с которыми проводились эксперименты
- `List<double> result` - результаты экспериментов
- `accuracy` - точность поиска (по умолчанию 0.0001)
- `lambda` - начальное значение шага (по умолчанию 0.0001)

Во время инициализации нового экземпляра класса происходит построение функционала (5.2). Это делается путем замены всех константных узлов новыми переменными. Начальные значения констант сохраняются в списке `InConstant`.

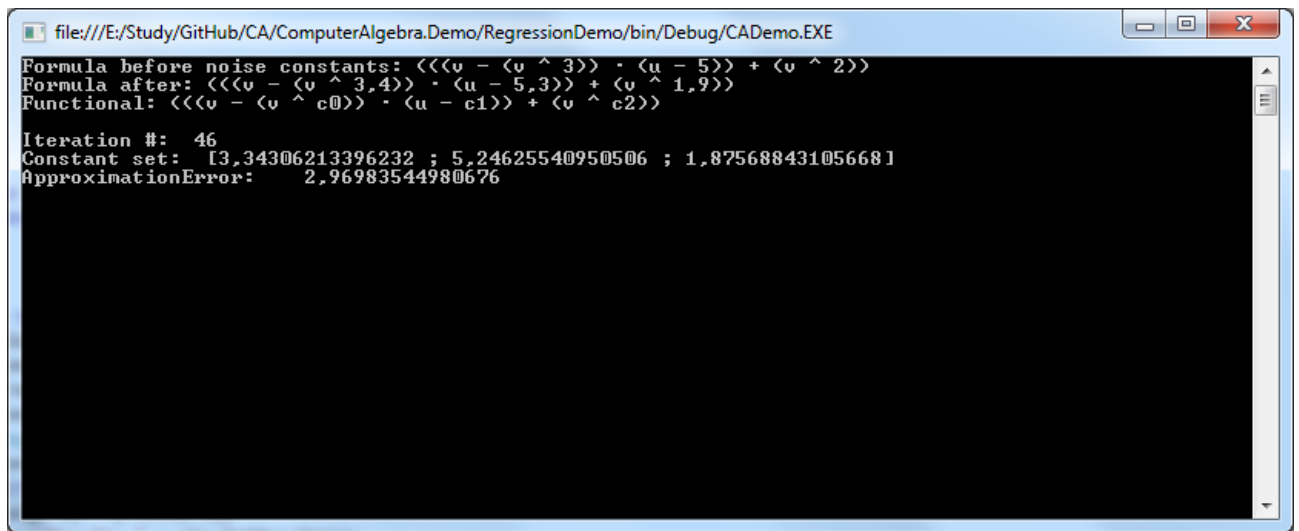
Для запуска алгоритма необходимо вызвать метод `Run`. Он принимает всего один необязательный параметр - число итераций. По умолчанию алгоритм заканчивает работу, если число итераций превысило максимально допустимое значение (1000), либо функция H из (5.4) для текущего набора констант меньше, чем заданная точность поиска.

Дальнейшая работа алгоритма заключается в последовательном вычислении градиента функции H (функция `MakeGradientStep`), изменении переменных в соответствии с (5.6) и пересчете шага по формулам (5.7) и (5.8) соответственно (функция `MakeIteration`). Неочевидность реализации остается только в том, как вычислять градиент. Распишем, чему равна частная производная функции H по переменной a_k :

$$\frac{\partial H}{\partial a_k} = \sum_{i=1}^{|M|} 2(G(x_{i1}, \dots, x_{in}, a_1, \dots, a_n) - z_i) \frac{\partial G(x_{i1}, \dots, x_{in}, a_1, \dots, a_n)}{\partial a_k}$$

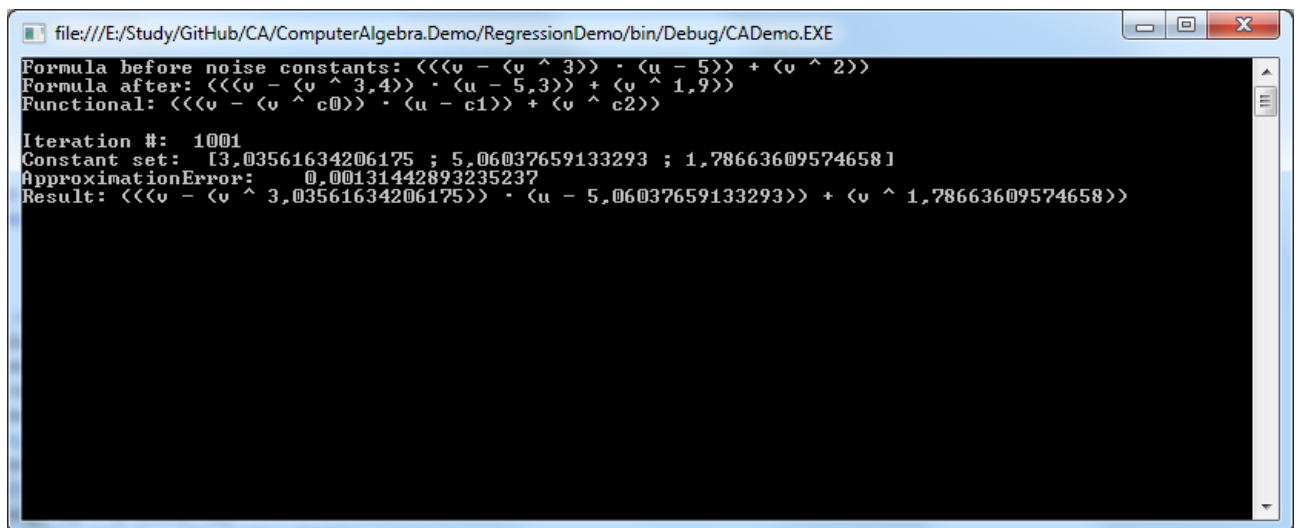
Функция G - это функция, построенная при инициализации класса. Используя символьное дифференцирование из раздела 4, нахождение частной производной этой функции не составляет никакого труда.

Результаты работы алгоритма можно видеть на рисунках ниже.



```
file:///E:/Study/GitHub/CA/ComputerAlgebra.Demo/RegressionDemo/bin/Debug/CADemo.EXE
Formula before noise constants: <<<v - <v ^ 3>> - <u - 5>> + <v ^ 2>>
Formula after: <<<v - <v ^ 3,4>> - <u - 5,3>> + <v ^ 1,9>>
Functional: <<<v - <v ^ c0>> - <u - c1>> + <v ^ c2>>
Iteration #: 46
Constant set: [3,34306213396232 ; 5,24625540950506 ; 1,87568843105668]
ApproximationError: 2,96983544980676
```

Рис. 5.1: Промежуточный результат работы алгоритма на 46 итерации



```
file:///E:/Study/GitHub/CA/ComputerAlgebra.Demo/RegressionDemo/bin/Debug/CADemo.EXE
Formula before noise constants: <<<v - <v ^ 3>> - <u - 5>> + <v ^ 2>>
Formula after: <<<v - <v ^ 3,4>> - <u - 5,3>> + <v ^ 1,9>>
Functional: <<<v - <v ^ c0>> - <u - c1>> + <v ^ c2>>
Iteration #: 1001
Constant set: [3,03561634206175 ; 5,06037659133293 ; 1,78663609574658]
ApproximationError: 0,00131442893235237
Result: <<<v - <v ^ 3,03561634206175>> - <u - 5,06037659133293>> + <v ^ 1,78663609574658>>
```

Рис. 5.2: Остановка алгоритма после 1000 итераций и подстановка
вычисленных значений

6 Правило резолюций

6.1 Построение дерева выражений для логики первого порядка

Язык логики первого порядка строится на основе сигнатуры, состоящей из множества функциональных символов \mathcal{F} и множества предикатных символов \mathcal{P} . С каждым функциональным и предикатным символом связана арность, то есть число возможных аргументов. Допускаются как функциональные, так и предикатные символы арности 0. Первые иногда выделяют в отдельное множество *констант*. Кроме того, используются следующие дополнительные символы:

- Символы переменных (обычно строчные буквы из конца латинского алфавита)
- Пропозициональные связки: $\vee, \wedge, \neg, \rightarrow, \leftrightarrow$
- Кванторы: всеобщности \forall и существования \exists
- Служебные символы: скобки и запятая.

Перечисленные символы вместе с символами из \mathcal{F} и \mathcal{P} образуют *Алфавит* логики первого порядка. Более сложные конструкции делятся на термы, атомы и формулы.

Термы определяются рекурсивно следующим образом:

- Константа есть терм.
- Переменная есть терм.
- Если f - n -местный функциональный символ и t_1, \dots, t_n - термы, то $f(t_1, \dots, t_n)$ - терм.
- Никаких термов, кроме порожденных применением указанных выше правил, нет.

Атом имеет вид $P(t_1, \dots, t_n)$, где P - предикатный символ арности n , а t_1, \dots, t_n - термы.

Формулы, как и термы, определяются рекурсивно:

1. Атом есть формула.
2. Если F и G - формулы, то $\neg F, F \vee G, F \wedge G, F \rightarrow G$ и $F \leftrightarrow G$ - формулы
3. Если F - формула, а x - свободная переменная в F , то $(\forall x)F$ и $(\exists x)F$ - формулы.
4. Формулы порождаются только конечным числом применений правил (1), (2) и (3) [8].

Для правила резолюций необходимо иметь возможность задавать бескванторную часть формул в сколемовской нормальной форме. Если посмотреть на структуру системы, описанную в главе 1, то можно увидеть, какие конструкции логики реализованы. Однако задавать формулы последовательным вызовом конструкторов и статических методов не очень удобно, кроме того необходимо получить единообразный способ задания логических и алгебраических выражений. Для достижения этой цели были сгенерированы специальные методы для некоторого числа предикатов, функций и констант, что позволило задавать формулы в виде лямбда-выражений:

$$(x, y, z) \Rightarrow P(f(x), g(y)) \mid !Q(a, h(z)) \mid R(z)$$

6.2 Алгоритм унификации

Прежде чем применять правило резолюций к двум формулам, их нужно унифицировать. Существует много алгоритмов, которые можно использовать для унификации конечного множества унифицируемых выражений и которые сообщают о неудаче, когда это множество не может быть унифицируемо. В системе реализован собственный алгоритм унификации, описанный ниже.

За унификацию в системе отвечает сервис `UnificationService`. В нем определены следующие статические методы:

- **GetUnificationRules** - получение правил унификации для двух сколемовских предикатов
- **Unificate** - унификация предиката по переданным правилам
- **CanUnificate** - проверка, что два предиката можно унифицировать.

Процедура **GetUnificationRules**, приводимая неформально, демонстрирует, каким образом происходит поиск наиболее общего унификатора.

```

1: procedure GETUNIFICATIONRULES( $P1, P2$ )
2:   if  $P1 \neq P2$  then
3:     return FAIL
4:   end if
5:    $rules \leftarrow \{\}$ 
6:   for  $i \leftarrow 1$  to  $\text{арность } P1$  do
7:      $F1 \leftarrow i \text{ элемент } P1$ 
8:      $F2 \leftarrow i \text{ элемент } P2$ 
9:     if  $F1, F2$  идентичны then
10:      continue
11:     else if  $F1$  переменная then
12:        $rules.Add(\{F2/F1\})$ 
13:     else if  $F2$  переменная then
14:        $rules.Add(\{F1/F2\})$ 
15:     else if  $F1 == F2$  then
16:        $Z1 \leftarrow \text{GetUnificationRules}(F1, F2)$ 
17:       if  $Z1 == \text{FAIL}$  then
18:         return FAIL
19:       end if
20:        $rules.Add(Z1)$ 
21:     else
22:       return FAIL
23:     end if
24:    $\text{Unificate}(P1, rules)$ 
25:    $\text{Unificate}(P2, rules)$ 

```

```

26:   end for
27: end procedure

```

6.3 Задание правила резолюции

Правило резолюции можно сформулировать так:

Для любых двух дизъюнктов C_1 и C_2 , если существует литера L_1 в C_1 , которая контрарна литере L_2 в C_2 , то вычеркнув L_1 и L_2 из C_1 и C_2 соответственно, построим дизъюнкцию оставшихся дизъюнктов. Построенный дизъюнкт есть резольвента C_1 и C_2 [8]

Пример:

$$C_1 : P \vee R$$

$$C_2 : !P \vee Q$$

Дизъюнкт C_1 имеет литеру P , которая контрарна литере $!P$ в C_2 . Следовательно, вычеркивая P и $!P$ из C_1 и C_2 соответственно и составляя дизъюнкцию оставшихся дизъюнктов R и Q , получаем резольвенту $R \vee Q$.

Благодаря тому, что формулы задаются в виде синтаксического дерева, правило резолюций можно задать стандартным правилом системы из раздела 2:

Rule

```

.New("Resolve", StdTags.Inductive, StdTags.Logic,
    StdTags.SafeResection)
.Select(A[ChildB], C[ChildD])
.Where<Logic.MultipleOr, SkolemPredicateNode, Logic.MultipleOr,
    SkolemPredicateNode>
    (z => UnificationService.CanUnificate(z.B, z.D)
    && (z.B.IsNegate || z.D.IsNegate))
.Mod(Modifier);

```

Листинг 6.1: Правило резолюций

Функция `Modifier` производит унификацию узлов и удаление контрарных предикатов.

Для получения резольвенты дизъюнктов, достаточно вызвать метод `Resolve` из сервиса `ComputerAlgebra`. В некоторых ситуациях контрарных

предикатов может быть несколько. Например, для дизъюнктов $P(x) \vee !Q(x) \vee R(x)$ и $!P(x) \vee Q(x) \vee !R(x)$ резольвентой может быть 3 различных дизъюнкта. Именно поэтому метод принимает два дизъюнкта и возвращает список резольвент.

Пример применения правила резолюций можно видеть на рисунке:

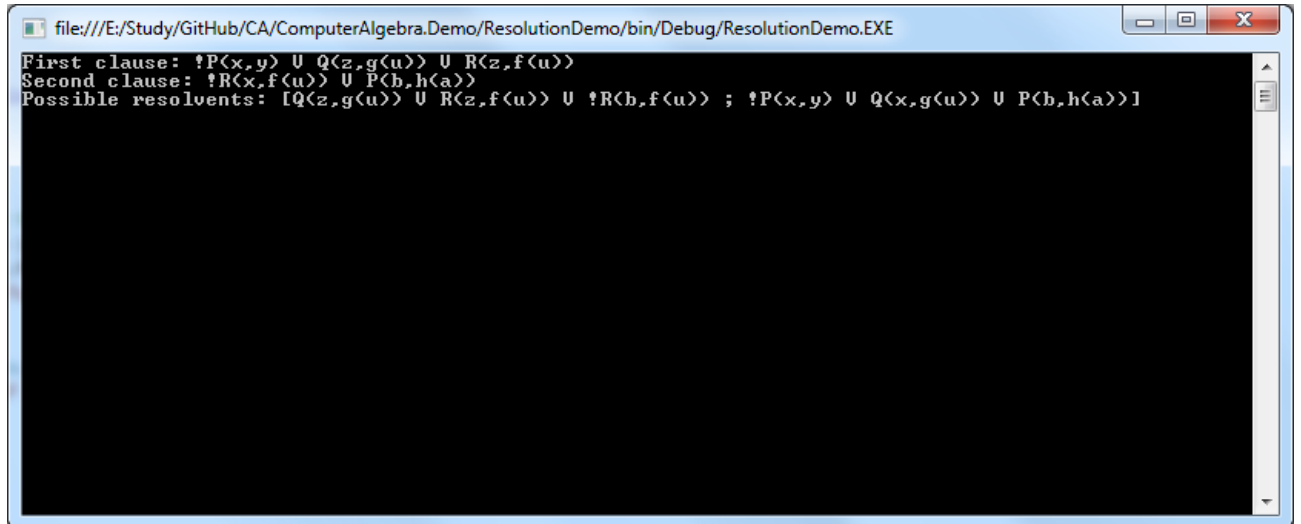


Рис. 6.1: Пример применения правила резолюций

ЗАКЛЮЧЕНИЕ

Конечным результатом работы стало создание работающей системы компьютерной алгебры, имеющей как практическое, так и методологическое значение. Система имеет огромный потенциал - каждая рассмотренная здесь функциональность может быть улучшена и расширена в дальнейшем. Так, например, правило резолюций играет главную роль в методе резолюций, который используется в большом числе задач.

Таким образом, подводя итог всему вышесказанному, можно утверждать, что работа по данной тематике может продолжаться в двух основных направлениях:

- Расширение существующего функционала
- Реализация новых направлений исследования - символьное интегрирование, алгебраическое решение дифференциальных уравнений и другие задачи компьютерной алгебры.

Истинная практическая ценность проекта проявляется тогда, когда он используется кем то, кроме разработчика. Поэтому проект был выложен в сообществе *www.codeproject.com* и в публичном репозитории *www.github.com*. Возможно он найдет свою аудиторию и будет продолжать развиваться без участия первоначальных авторов.

ЛИТЕРАТУРА

- [1] .NET Development. <http://msdn.microsoft.com/en-us/library/ff361664.aspx>
- [2] Панов Н.В., *Разработка рандомизированных алгоритмов в интервальной глобальной оптимизации*. - Новосибирск, 2012
- [3] Бухбергер Б., Калме Ж., Калтофен Э., Коллинз Дж., Лауэр М., Лафон Ж., Лоос Р., Минньотт М., Нойбюзер Й., Норман А., Уинклер Ф., ван Хюльзен Я. *Компьютерная алгебра: Символьные и алгебраические вычисления*. М.: Мир, 1996.
- [4] Хювёнен Э., Сеппянен Й. *Мир Лиспа. В 2-х томах*. Пер.с англ. - М.: Мир, 1990
- [5] Дрейпер Н., Смит Г. *Прикладной регрессионный анализ. Второе издание*. Пер.с англ. - М.: Финансы и статистика, 1986
- [6] Метод наименьших квадратов (расчёт коэффициентов)
[http://ru.wikipedia.org/wiki/Регрессионный анализ](http://ru.wikipedia.org/wiki/Регрессионный_анализ)
- [7] Базара М., Шетти К. *Нелинейное программирование. Теория и алгоритмы*. Пер.с англ. – М.:Мир, 1982, - 563 с.
- [8] Чень Ч., Ли Р. *Математическая логика и автоматическое доказательство теорем* — М.: Наука, 1983, — 358 с.

ПРИЛОЖЕНИЕ

```
(define (diff expr x)
  (define (dx t)
    (if (pair? t)
        (if (null? (cddr t))
            (unary (car t) (cadr t)))
            (binary (car t) (cadr t)(caddr t))))
    (if (eq? t x) 1 0)))
(define (binary f u v)
  (cond
    ((eq? f '+) (list '+ (dx u) (dx v)))
    ((eq? f '-') (list '- (dx u) (dx v)))
    ((eq? f '*') (list '+
                        (list '* (dx u) v)
                        (list '* (dx v) u)))
    ((eq? f '/') (list '/
                        (list '-
                          (list '* (dx u) v)
                          (list '* (dx v) u))
                        (list 'expt v 2)))
    ((eq? f 'expt)(list '+
                        (list '*
                          (dx u) v
                          (list 'expt u (list '- v 1)))
                        (list '*
                          (dx v)
                          (list 'expt u)
                          (list 'log u))))
    (else (error "unknown operatrion"))))
(define (unary f u)
  (list '*
        (dx u)
        (cond ((eq? f 'exp) (list 'exp u))
              ((eq? f 'log) (list '/ u))
              ((eq? f 'sin) (list 'cos u))
              ((eq? f 'cos) (list '- (list 'sin u)))
              (else (error "unknown operatrion"))))
```

```
(dx expr))
```

Листинг 1: Код алгоритма символьного дифференцирования [2]