Тверской государственный технический университет
Факультет информационных технологий
Программное обеспечение вычислительной техники

Структуры и алгоритмы обработки данных

Курсовая работа
Алгоритмы вычислительной математики

Работу выполнил:
А. В. Малов
Группа:
Б.ПИН.РИС-20.05
Преподаватель:
А. А. Мальков

Тверь
2022

Содержание

Введение

В данной курсовой работе будут показаны возможности языка `C#` в плане использования его, как функционального языка программирования. В частности, будет сравнение алгоритмов вычисления одной и той же функции применяя разные парадигмы программирования. Сравним оптимизацию хвостовой рекурсией, обычную рекурсию, императивный стиль (цикл), а так же воспользуемся стандартными средствами языка `C#` для того, чтобы вычислить функцию, используя все ядра нашего процессора.

По сей день люди решают уравнения (находят корни уравнения), находят суммы или произведения рядов, поэтому данная работа является актуальной. Так же эта работа является в некотором роде учебной, так как исследуется то, что не было в курсе программирования.

Цель данной работы - изучение основ функциональной парадигмы программирования, применение принципов функционального программирования с помощью языка `C#`.

Задачи:

- Изучить литературу по данной теме.
- Изучить методы и алгоритмы решения задачи.
- Описать сложность для каждого представленного алгоритма в O-нотации.

1 Функциональная парадигма программирования

1.1 История

Первый высокоуровневый функциональный язык программирования `LISP` был разработан в конце 1950-х годов для научных компьютеров серии `IBM 700/7000` Джоном Маккарти, когда он учился в Массачусетском технологическом институте (MIT). [2]

Язык обработки информации (`IPL`), 1956, иногда упоминается как первый функциональный язык программирования. Это язык в стиле ассемблера для работы со списками символов. [3] В нём действительно есть понятие генератора, которое сводится к функции, которая принимает функцию в качестве аргумента, и, поскольку это язык уровня ассемблера, то код может быть данными, поэтому IPL можно рассматривать как язык, имеющий функции более высокого порядка.

В 1970-х годах Гай Л. Стил и Джеральд Джей Сассман разработали язык `Scheme`, и использовали его в своём курсе, основанном на книге 1985 года "Структура и интерпретация компьютерных программ"[1]. `Scheme` был первым диалектом `LISP`, который использовал лексическую область видимости и требовал оптимизацию хвостовой рекурсией, которые рекомендуются в функциональном программировании. Книга обучает фундаментальными принципами компьютерного программирования, включая рекурсию, абстракцию, модульность, проектирование и реализация языков программирования. Именно из этой книги были взяты некоторые примеры. Книга до сих поря является классикой жанра и её до сих пор преподают в самых известных высших заведениях США.

1.2 Функциональное программирование

Функциональное программирование - это:

- парадигма программирования, характеризующаяся использованием математических функций и избеганием побочных эффектов.
- стиль программирования, который использует только чистые функции без побочных эффектов.

Побочные эффекты - это всё, что делает функция, кроме возврата значения, например:

1. Отправка электронного письма.
2. Чтение файла.
3. Выполнение веб-запроса.

Побочные эффекты приводят к тому, что программист не может предсказать, что произойдёт в его программе.

Чистые функции - это функции, которые зависят только от своих аргументов и не имеют никаких побочных эффектов. Одни и те же аргументы всегда будут выдавать одно и то же возвращаемое значение.

Из определения можно подумать, что функциональные языки программирования не имеют грязных функций. Но это не так - такой язык бесполезен в реальном мире. Вход в мир грязных функций красиво реализован в языке Haskell, там для таких целей используются монады, с помощью которых мы делаем обёртку над грязным миром.

1.3 Концепции

В целом можно выделить несколько инструментов (концептов) функционального программирования, это:

- Функции первого класса и функции высшего порядка.
- Чистые функции.
- Рекурсия.
- Строгая и нестрогая вычислительная система.

1.3.1 Функции высшего и первого порядка

В математике и компьютерных науках функция высшего порядка - это функция, которая выполняет по крайней мере одно из следующих действий:

- принимает одну или несколько функций в качестве аргументов,
- возвращает функцию в качестве результата.

Все остальные функции являются функциями первого класса.

1.3.2 Чистые функции

Чистые функции (или выражения) не имеют побочных эффектов (прямая работа с памятью или вводом-выводом). Это означает, что чистые функции обладают несколькими полезными свойствами, многие из которых могут быть использованы для оптимизации кода:

- Если результат чистого выражения не используется, его можно удалить, не затрагивая другие выражения.
- Если вызывается чистая функция с аргументами, которые не вызывают побочных эффектов, результат остается постоянным по отношению к этому списку аргументов (иногда вызывается ссылочная прозрачность или идемпотентность), т.е. повторный вызов чистой функции с теми же аргументами возвращает тот же результат. (Это может быть использовано для оптимизацией кэшированием)
- Если между двумя чистыми выражениями нет зависимости от данных, их порядок может быть изменен на противоположный, или они могут выполняться параллельно, и они не могут мешают друг другу (другими словами, вычисление любого чистого выражения является потокобезопасным).
- Если весь язык не допускает побочных эффектов, то можно использовать любую стратегию вычисления (следующая концепция, которую мы рассмотрим); это дает компилятору свободно изменять порядок или комбинировать вычисление выражений в программе (компилятор может так применить дефорестацию - избегание построения новых структур данных в программе).

1.3.3 Рекурсия

Итерация (цикл) в функциональных языках обычно выполняется с помощью рекурсии. Рекурсивные функции вызывают сами себя, позволяя повторять операцию до тех пор, пока она не достигнет базового варианта. В общем случае рекурсия требует поддержания стека, который потребляет пространство в линейной зависимости от глубины рекурсии. Это может сделать рекурсию непомерно дорогой для использования вместо императивных циклов. Однако особая форма рекурсии, известная как хвостовая рекурсия, может быть распознана и оптимизирована компилятором в том же коде, который использовался для реализации итерация в императивных языках. Оптимизация хвостовой рекурсии может быть реализована, среди прочего, путем преобразования программы в стиль передачи продолжения во время компиляции.

1.3.4 Стратегия вычисления

Функциональные языки можно классифицировать по тому, используют ли они строгое (незамедлительное) или нестрогое (ленивое) вычисление, это понятия, которые относятся к тому, как аргументы функций вычисляются при вычислении самой функции.

Например:

```
print(len([2+1, 3*2, 1/0, 5-4]))
```

Данное выражение не проходит строгое вычисление из-за деления на ноль в третьем элементе списка. При ленивом вычислении функция длины возвращает значение 4 (т.е. количество элементов в списке), поскольку при ее вычислении не предпринимается попытка вычисления внутренних выражений, составляющие список. Короче говоря, строгая оценка всегда полностью оценивает аргументы функции перед вызовом функции. Ленивая оценка не вычисляет аргументы функции, если только их значения не требуются для вычисления самой функции.

2 Выбор программного обеспечения

2.1 Язык программирования

Так как идея разработки пришла из прочтения книг про функциональное программирование, то и было бы логично применять в данной работе языки, которые являются либо частично функциональными, либо чисто функциональными.

Но одно из требований написания курсовой работы - применять язык программирования `C#`, который не является функциональным. В нём есть инструменты, с помощью которых можно писать функциональные методы, но это приходится всё обёртывать в статические классы, которые служат собой обычным именованным пространством.

Из-за требований работы невозможно рассматривать альтернативу `C#`.

На данный момент существует функциональный язык программирования `F#`, который может запускать `C#` код.

2.2 Операционная система

На сегодняшний день почти все языки программирования имеют кроссплатформенную поддержку почти всех популярных операционных систем (`BSD`, `Windows`, `Linux`, `MacOS`).

Разработка велась на операционной системе `Windows 10`.

2.3 Инструменты

В качестве инструмента написания программ было выбрано несколько программ:

- `Microsoft Visual Studio`
- `Microsoft Visual Studio Code`
- `JetBrains Rider`

`Microsoft Visual Studio` поставляет компилятор и интерпретатор для языка `C#` под операционную систему `Windows`.

`Microsoft Visual Studio Code` является легковесным редактором текста. В нём есть удобные расширения, с помощью которых я исправлял грамматические ошибки в комментариях к каждому методу.

`JetBrains Rider` является моим основным средством разработки. Платный, но для студентов предоставляется студенческая бесплатная лицензия. В этой среде разработки имеется удобный дебаггер, статический анализатор кода, который помогает найти ошибки в коде. Удобные подсказки и автозаполнение кода увеличивают скорость и качество разработки кода.

В качестве системы контроля версий был выбран `Git`. Является стандартом в современной разработке софта. Весь текст курсовой работы и сам проект находится на сервисе `GitHub` (`https://github.com/andreymlv/coursework-functional-programming-in-charp`).

3 Проектирование

При проектировании были использованы некоторые аспекты гибкой методологии разработки программного обеспечения.

В частности, была установлена основная функциональность проекта:

1. Разработка функций высшего порядка, которые вычисляли сумму или произведение рядов любых функций.
2. Разработка двух методов решения уравнений (метод Ньютона и метод бисекций).
3. Разработка оптимизации хвостовой рекурсией.

Разработка функций высшего порядка не является трудоёмкой задачей. Программисту необходимо сделать обёртку над какой-то общей идеей, чтобы пользователь данного кода думал об этих функциях, как о неком интерфейсе взаимодействия.

Например, вычисление суммы ряда. В данной курсовой работе есть целых 4 стратегии вычисления. Эти стратегии и будут аргументами, которые будут передаваться в параметры функции вычисления суммы некого ряда. В нашем примере это будет: обычная рекурсия, "рекурсия"с использованием оптимизации хвостовой рекурсией, обычный цикл, и с использованием стандартных средств `DotNet` распараллеливание.

Или, вычисление корней уравнения: в данной работе показано, как можно использовать метод нахождения неподвижной функции для поиска корня уравнения. Проблема в том, что не каждая функция сходится, когда мы ищем неподвижную точку. Поэтому нам нужны дополнительные стратегии, или, назовём их иначе - помощники, которые помогут вычислить эту самую неподвижную точку функции. В нашем примере это будет: метод касательных (или метод Ньютона) и метод торможения усреднением.

Оптимизация хвостовой рекурсии является задачей компилятора. Но не все компиляторы поддерживают данную возможность из-за того, что язык первоначально не был спроектирован для того, чтобы использовать рекурсию, как механизм цикла. В `C#` данную оптимизацию можно реализовать с помощью `trampoline`, это фрагмент кода, который многократно вызывает функции. Когда функция должна выполнить вызов другой функции, вместо того, чтобы вызывать ее напрямую, а затем возвращать результат, она возвращает адрес функции и параметры вызова возвращаются к `trampoline` (из которого он был вызван сам), и `trampoline` заботится о следующем вызове этой функции с заданными параметрами. Это гарантирует, что стек не растет и цикл может продолжаться бесконечно.

3.1 Трансформация кода в цикл

Хвостовая рекурсия может быть связана с оператором `while`, явной итерацией, например, путем преобразования:

```
def foo(x):
  if p(x):
    return bar(x)
  else:
    return foo(baz(x))
```

в

```
def foo(x):
  while true:
    if p(x):
      return bar(x)
    else:
      x = baz(x)
```

3.2 Алгоритм вычисления ряда

По-сути алгоритм такой: в цикле накапливаем результат. Примитивный алгоритм имеет сложность $O(n)$.

3.3 Численные методы решения нелинейных уравнений

3.3.1 Метод бисекции или метод деления отрезка пополам

Метод бисекции или метод деления отрезка пополам — простейший численный метод для решения нелинейных уравнений вида $f(x) = 0$. Предполагается только непрерывность функции $f(x)$. Поиск основывается на теореме о промежуточных значениях.

Задача заключается в нахождении корней нелинейного уравнения $f(x) = 0$.

Для начала итераций необходимо знать отрезок $[x_L, x_R]$, значений x, на концах которого функция принимает значения противоположных знаков.

Противоположность знаков значений функции на концах отрезка можно определить множеством способов. Один из множества этих способов — умножение значений функции на концах отрезка и определение знака произведения путём сравнения результата умножения с нулём: $f(x_L) * f(x_R) < 0$, в действительных вычислениях такой способ проверки противоположности знаков при крутых функциях приводит к преждевременному переполнению. Для устранения переполнения и уменьшения затрат времени, то есть для увеличения быстродействия, на некоторых программно-компьютерных комплексах противоположность знаков значений функции на концах отрезка нужно определять по формуле: $sign(f(x_L)) \neq sign(f(x_R))$, так как одна операция сравнения двух знаков двух чисел требует меньшего времени, чем две операции: умножение двух чисел (особенно с плавающей запятой и двойной длины) и сравнение результата с нулём. При данном сравнении, значения функции $f(x)$ в точках x_L и x_R можно не вычислять, достаточно вычислить только знаки функции $f(x)$ в этих точках, что требует меньшего машинного времени.

Из непрерывности функции $f(x)$ и условия противоположности знаков следует, что на отрезке $[x_L, x_R]$ существует хотя бы один корень уравнения (в случае не монотонной функции $f(x)$ функция имеет несколько корней и метод приводит к нахождению одного из них).

Найдём значение x в середине отрезка:$x_M = (x_L + x_R)/2$, в действительных вычислениях, для уменьшения числа операций, в начале, вне цикла, вычисляют длину отрезка по формуле:$x_D = (x_R - x_L)$, а в цикле вычисляют длину очередных новых отрезков по формуле: $x_D = x_D/2$ и новую середину по формуле:$x_M = x_L + x_D$.

Вычислим значение функции $f(x_M)$ в середине отрезка x_M:

Если $f(x_M) = 0$ или, в действительных вычислениях, $|f(x_M)| \leq \varepsilon_{f(x)}$, где $\varepsilon_{f(x)}$ - заданная точность по оси y, то корень найден. Иначе $f(x_M) \neq 0$ или, в действительных вычислениях, $|f(x_M)| > \varepsilon_{f(x)}$, то разобьём отрезок $[x_L, x_R]$ на два равных отрезка: $[x_L, x_M]$ и $[x_M, x_R]$.

Теперь найдём новый отрезок, на котором функция меняет знак:

- Если значения функции на концах отрезка имеют противоположные знаки на левом отрезке, $f(x_L) \cdot f(x_M) < 0$ или $sign(f(x_L)) \neq sign(f(x_M))$, то, соответственно, корень находится внутри левого отрезка $[x_L, x_M]$. Тогда возьмём левый отрезок присвоением $x_R = x_M$, и повторим описанную процедуру до достижения требуемой точности $\varepsilon_{f(x)}$ по оси y.

- Иначе значения функции на концах отрезка имеют противоположные знаки на правом отрезке, $f(x_M) \cdot f(x_R) < 0$ или $sign(f(x_M)) \neq sign(f(x_R))$, то, соответственно, корень находится внутри правого отрезка $[x_M, x_R]$. Тогда возьмём правый отрезок присвоением $x_L = x_M$, и повторим описанную процедуру до достижения требуемой точности $\varepsilon_{f(x)}$ по оси y.

За количество итераций N деление пополам осуществляется N раз, поэтому длина конечного отрезка в 2^N раз меньше длины исходного отрезка. Получаем сложность $O(\log N)$.

3.3.2 Неподвижная точка и метод Ньютона

Неподвижная точка в математике — точка, которую заданное отображение переводит в неё же, иными словами, решение уравнения $f(x) = x$.

К примеру, отображение $f(x) = x^2 - 3x + 3$ имеет неподвижные точки $x = 1$ и $x = 3$, поскольку $f(1) = 1$ и $f(3) = 3$.

Неподвижные точки есть не у всякого отображения — скажем, отображение $f(x) = x+1$ вещественной прямой в себя неподвижных точек не имеет.

Точки, возвращающиеся в себя после определённого числа итераций, то есть, решения уравнения $f(f(\dots f(x) \dots)) = x$, называются периодическими (в частности, неподвижные точки — это периодические точки периода 1).

Одним из применений идеи притягивающей неподвижной точки является метод Ньютона: решение уравнения оказывается притягивающей неподвижной точкой некоторого отображения, и потому может быть найдено как предел очень быстро сходящейся последовательности чисел, полученных его повторным применением.

Основная идея метода заключается в следующем: задаётся начальное приближение вблизи предположительного корня, после чего строится касательная к графику исследуемой функции в точке приближения, для которой находится пересечение с осью абсцисс. Эта точка берётся в качестве следующего приближения. И так далее, пока не будет достигнута необходимая точность.

В этом случае алгоритм нахождения численного решения уравнения $f(x) = 0$ сводится к итерационной процедуре вычисления: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. По теореме Банаха последовательность приближений стремится к корню уравнения $f(x) = 0$.

Наиболее известным примером применения этого метода является нахождение квадратного корня из числа $a \geq 0$ как предела итераций отображения $f(x) = \dfrac{x + \frac{a}{x}}{2}$.

4 Разработка

Первым делом будет показан интерфейс взаимодействия, а после будет показано и рассказаны детали реализации.

Для демонстрации мощности функционального программирования я создал статический класс Accumulate, который является модулем, в котором есть функции нахождения суммы/произведения рядов.

```
namespace Library;

/// <summary>
///     Модуль накопления.
///     Четыре способа вычисления ряда:
///     - Рекурсивное вычисление.
///     - Рекурсивное вычисление с помощью оптимизации хвостовой рекурсией.
///     - Императивное вычисление.
///     - Параллельное вычисление.
/// </summary>
public static class Accumulate
{
    /// <summary>
    ///     Рекурсивный вызов оператора последовательности.
    ///     Возможно переполнение стека.
    /// </summary>
    /// <param name="combiner">Функция, которая будет применяться между двумя
    ↪  элементами последовательности.</param>
    /// <param name="term">Функция, которая будет применена к каждому элементу
    ↪  последовательности.</param>
    /// <param name="start">Начало отрезка.</param>
    /// <param name="next">Функция выбора следующего элемента, основываясь на
    ↪  предыдущем.</param>
    /// <param name="end">Конец отрезка.</param>
    /// <param name="based">Основа последовательности.</param>
    /// <returns>
    ///     Вычисление последовательности от <paramref name="start" /> до
    ↪  <paramref name="end" /> с шагом
    ///     <paramref name="next" /> с применением <paramref name="term" />
    ↪  для каждого элемента.
    /// </returns>
    public static double RecursiveInvoke(Func<double, double, double>
    ↪  combiner, Func<double, double> term, double start,
        Func<double, double> next, double end, double based)
    {
        return start > end
            ? based
            : combiner(term(start), RecursiveInvoke(combiner, term,
            ↪  next(start), next, end, based));
    }

    /// <summary>
```

```
///     Рекурсивный вызов оператора последовательности, применяя
 ↪  оптимизацию хвостовой рекурсии.
///     Утверждается, что код переобразуется в функцию вида <see
 ↪  cref="TailRecursiveInvoke" />
///     и теоретически соотвествовать функции <see cref="ImperativeInvoke"
 ↪  />
/// </summary>
/// <param name="combiner">Функция, которая будет применяться между двумя
 ↪  элементами последовательности.</param>
/// <param name="term">Функция, которая будет применена к каждому элементу
 ↪  последовательности.</param>
/// <param name="start">Начало отрезка.</param>
/// <param name="next">Функция выбора следующего элемента, основываясь на
 ↪  предыдущем.</param>
/// <param name="end">Конец отрезка.</param>
/// <param name="based">Основа последовательности.</param>
/// <returns>
///     Вычисление последовательности от <paramref name="start" /> до
 ↪  <paramref name="end" /> с шагом
///     <paramref name="next" /> с применением <paramref name="term" />
 ↪  для каждого элемента.
/// </returns>
public static double TailRecursiveInvoke(Func<double, double, double>
 ↪  combiner, Func<double, double> term,
    double start,
    Func<double, double> next, double end, double based)
{
    return TailRecursion.Execute(() => RecursiveInvokeHelper(combiner,
     ↪  term, start, next, end, based));
}

/// <summary>
///     Определение самой функции сумма с использованием аккумулятора.
/// </summary>
/// <param name="combiner">Функция, которая будет применяться между двумя
 ↪  элементами последовательности.</param>
/// <param name="term">Функция, которая будет применена к каждому элементу
 ↪  последовательности.</param>
/// <param name="start">Начало отрезка.</param>
/// <param name="next">Функция выбора следующего элемента, основываясь на
 ↪  предыдущем.</param>
/// <param name="end">Конец отрезка.</param>
/// <param name="based">Основа последовательности.</param>
/// <returns>
///     Вычисление последовательности от <paramref name="start" /> до
 ↪  <paramref name="end" /> с шагом
///     <paramref name="next" /> с применением <paramref name="term" />
 ↪  для каждого элемента.
/// </returns>
```

```
private static RecursionResult<double> RecursiveInvokeHelper(Func<double,
↪  double, double> combiner,
    Func<double, double> term, double start, Func<double, double> next,
    ↪  double end, double based)
{
    return start > end
        ? TailRecursion.Return(based)
        : TailRecursion.Next(() =>
            RecursiveInvokeHelper(combiner, term, next(start), next, end,
            ↪  combiner(based, term(start))));
}

/// <summary>
///     Императивный вызов оператора суммы.
/// </summary>
/// <param name="combiner">Функция, которая будет применяться между двумя
↪  элементами последовательности.</param>
/// <param name="term">Функция, которая будет применена к каждому элементу
↪  последовательности.</param>
/// <param name="start">Начало отрезка.</param>
/// <param name="next">Функция выбора следующего элемента, основываясь на
↪  предыдущем.</param>
/// <param name="end">Конец отрезка.</param>
/// <param name="based">Основа последовательности.</param>
/// <returns>
///     Вычисление последовательности от <paramref name="start" /> до
↪  <paramref name="end" /> с шагом
///     <paramref name="next" /> с применением <paramref name="term" />
↪  для каждого элемента.
/// </returns>
public static double ImperativeInvoke(Func<double, double, double>
↪  combiner, Func<double, double> term,
    double start,
    Func<double, double> next, double end, double based)
{
    for (var i = start; i <= end; i = next(i))
        based = combiner(based, term(i));
    return based;
}

/// <summary>
///     Параллельное вычисление последовательности с помощью стандартных
↪  средств DotNet.
/// </summary>
/// <param name="combiner">Функция, которая будет применяться между двумя
↪  элементами последовательности.</param>
/// <param name="term">Функция, которая будет применена к каждому элементу
↪  последовательности.</param>
/// <param name="start">Начало отрезка.</param>
```

```
    /// <param name="next">Функция выбора следующего элемента, основываясь на
    ↪  предыдущем.</param>
    /// <param name="end">Конец отрезка.</param>
    /// <param name="based">Основа последовательности.</param>
    /// <returns>
    ///     Вычисление последовательности от <paramref name="start" /> до
    ↪  <paramref name="end" /> с шагом
    ///     <paramref name="next" /> с применением <paramref name="term" />
    ↪  для каждого элемента.
    /// </returns>
    public static double DotNetInvoke(Func<double, double, double> combiner,
    ↪  Func<double, double> term, double start,
        Func<double, double> next, double end, double based)
    {
        return ParallelEnumerable
            .Range((int) start, (int) (end - start) + 1) // Создаём отрезок
            ↪  [start; end] типа int
            .Where((_, i) => i % next(0) == 0) // Фильтруем элементы с шагом
            ↪  next(0)
            .Select<int, double>(i => i) // Конвертируем отрезок в double
            .Select(term) // Для каждого отфильтрованного элемента применяем
            ↪  функцию term
            .Aggregate(based, combiner);
        // Применяем оператор последовательности
    }
}
```

Листинг 1: `Accumulate.cs` - Статический класс для создания любых рядов.

Такой класс создаёт больше поле стратегий для вычислений любого ряда.

Можно задать любой шаг. Можно задать любую функцию, которая описывает общий элемент последовательности. Любой знак между общими членами. Любой аккумулятор, который будет накапливать значение.

Таким образом можно создать класс, который "реализует"некоторые части нашей последовательности. Например, класс "сумма ряда":

```
namespace Library;

/// <summary>
///     Реализация оператора суммы (сигма).
/// </summary>
public static class Sum
{
    private const double Base = 0.0d;

    private static double Operator(double a, double b)
    {
        return a + b;
    }

    /// <summary>
```

```
    ///     Вычисление суммы последовательности <paramref name="term" /> от
    ↪  <paramref name="start" />
    ///     до <paramref name="end" />, где следующий элемент выбирается с
    ↪  помощью <paramref name="next" />.
    ///     Стратегия вычисления выбирается с помощью функций в <see
    ↪  cref="Accumulate" />.
    /// </summary>
    /// <param name="strategy">Стратегия вычисления.</param>
    /// <param name="term">Общий член последовательности.</param>
    /// <param name="start">Начало последовательности.</param>
    /// <param name="next">Выбор следующего элемента.</param>
    /// <param name="end">Конец последовательности.</param>
    /// <returns>Вычисление последовательности с заданными
    ↪  параметрами.</returns>
    public static double Solve(
        Func<Func<double, double, double>, Func<double, double>, double,
        ↪  Func<double, double>, double, double, double>
            strategy, Func<double, double> term, double start, Func<double,
            ↪  double> next, double end)
    {
        return strategy(Operator, term, start, next, end, Base);
    }
}
```

Листинг 2: Sum.cs - Статический класс для создания любых рядов суммы.

Как видим мы используем функции высшего порядка. Мы передаём в качестве аргументов "стратегию"вычисления суммы. В юнит-тестах показано, как использовать данных класс. Аналогичным образом был разработал класс "произведения ряда":

```
namespace Library;

/// <summary>
///     Реализация оператора умножения (П).
/// </summary>
public static class Product
{
    private const double Base = 1.0d;

    private static double Operator(double a, double b)
    {
        return a * b;
    }

    /// <summary>
    ///     Вычисление произведения последовательности <paramref name="term"
    ↪  /> от <paramref name="start" />
    ///     до <paramref name="end" />, где следующий элемент выбирается с
    ↪  помощью <paramref name="next" />.
    ///     Стратегия вычисления выбирается с помощью функций в <see
    ↪  cref="Accumulate" />.
```

```
    /// </summary>
    /// <param name="strategy">Стратегия вычисления.</param>
    /// <param name="term">Общий член последовательности.</param>
    /// <param name="start">Начало последовательности.</param>
    /// <param name="next">Выбор следующего элемента.</param>
    /// <param name="end">Конец последовательности.</param>
    /// <returns>Вычисление последовательности с заданными
    ↪  параметрами.</returns>
    public static double Solve(
        Func<Func<double, double, double>, Func<double, double>, double,
        ↪  Func<double, double>, double, double, double>
            strategy, Func<double, double> term, double start, Func<double,
            ↪  double> next, double end)
    {
        return strategy(Operator, term, start, next, end, Base);
    }
}
```

Листинг 3: `Product.cs` - Статический класс для создания любых рядов произведения.

Эти классы показывают как можно легко построить систему, в основе которой лежит более общая "идея". Идея суммы ряда это частный случай любой последовательности.

Далее стоит обратить своё внимание на то, как реализована оптимизация хвостовой рекурсией, потому что в самом компиляторе и интерпретаторе кода C# нет такой оптимизации. Как было уже ранее показано в пункте 3.1, любая примитивная рекурсия может быть трансформирована в цикл.

Для того, чтобы передавать в trampoline нашу рекурсию, программисту необходимо определить функцию-помощника `RecursiveInvokeHelper`, которая являлась бы обёрткой той реализации, которую программист написал бы без оптимизации. Базовый случай рекурсии необходимо вызвать как `TailRecursion.Return(...)`, а общий случай необходимо вызывать как исполнение нашей функции, передав её, как лямбда-функцию `TailRecursion.Next(() => RecursiveInvokeHelper(...))`. Далее, чтобы начать исполнение нашей рекурсии нам надо вызвать код, который передаст все аргументы и функции в trampoline: `TailRecursion.Execute(() => RecursiveInvokeHelper(...))`.
И когда `Execute` начнёт своё выполнение, то он развернёт рекурсию в бесконечный цикл, который прервётся только при выдаче нашей рекурсивной функции базового случая. Вот сама реализация нашего trampoline:

```
namespace Library;

/// <summary>
///     Реализация оптимизации хвостовой рекурсии через лямбду.
/// </summary>
public static class TailRecursion
{
    /// <summary>
    ///     Запуск рекурсивной функции, превращая её в императивный стиль.
    /// </summary>
    /// <typeparam name="T">Тип данных, возвращаемой рекурсивной
    ↪  функции.</typeparam>
    /// <param name="func">Рекурсивная функция.</param>
```

```
    /// <returns>Результат выполнения рекурсии.</returns>
    public static T Execute<T>(Func<RecursionResult<T>> func)
    {
        while (true)
        {
            var (isFinalResult, result, nextStep) = func();
            if (isFinalResult)
                return result;
            func = nextStep;
        }
    }

    /// <summary>
    ///     Завершаем рекурсию, возвращаем результат выполнения.
    /// </summary>
    /// <typeparam name="T">Тип данных, возвращаемой рекурсивной
    ↪  функции.</typeparam>
    /// <param name="result">Сам результат.</param>
    /// <returns>Результат выполнения рекурсии.</returns>
    public static RecursionResult<T> Return<T>(T result)
    {
        return new RecursionResult<T>(true, result, null!);
    }

    /// <summary>
    ///     Выбираем следующий элемент в рекурсии.
    /// </summary>
    /// <param name="nextStep">Функция, которая выбирает следующий
    ↪  элемент.</param>
    /// <typeparam name="T">Тип данных, возвращаемой рекурсивной
    ↪  функции.</typeparam>
    /// <returns>Следующий элемент рекурсии.</returns>
    public static RecursionResult<T> Next<T>(Func<RecursionResult<T>>
    ↪  nextStep)
    {
        return new RecursionResult<T>(false, default!, nextStep);
    }
}
```

Листинг 4: `TailRecursion.cs` - Статический класс для оптимизации хвостовой рекурсии (trampoline).

И есть класс, который является контейнером для текущего состояния нашей рекурсии:

```
namespace Library;

/// <summary>
///     Контейнер для результата рекурсии.
/// </summary>
/// <typeparam name="T">Тип данных, которую должна вернуть рекурсивная
↪  функция</typeparam>
```

```
/// <param name="IsFinalResult">Получили ли мы конечный результат вычислений
 ↪  рекурсии?</param>
/// <param name="Result">Сам результат выполнения рекурсивной функции.</param>
/// <param name="NextStep">Функция, которая воспроизводит следующее значение
 ↪  для выполнения нашей рекурсивной функции.</param>
public readonly record struct RecursionResult<T>(bool IsFinalResult, T Result,
 ↪  Func<RecursionResult<T>> NextStep)
{
}
```

Листинг 5: `RecursionResult.cs` - Контейнер состояния рекурсии.

5 Тестирование

Были произведены тесты на основе фреймворка xUnit для юнит-тестирования:

```
using Library;
using Xunit;

namespace Tests;

public class SumUnitTest
{
    private static double Term(double x)
    {
        return x;
    }

    private static double Next(double x)
    {
        return x + 1;
    }

    [Theory]
    [InlineData(1, 1, 1)]
    [InlineData(1, 2, 3)]
    [InlineData(-5, 5, 0)]
    [InlineData(1, 10, 55)]
    public void RecursiveInvokeTest(double from, double to, double expected)
    {
        var actual = Sum.Solve(Accumulate.RecursiveInvoke, Term, from, Next,
        ↪  to);
        Assert.Equal(expected, actual);
    }

    [Theory]
    [InlineData(1, 1, 1)]
    [InlineData(1, 2, 3)]
    [InlineData(-5, 5, 0)]
    [InlineData(1, 10, 55)]
    public void TailRecursiveInvokeTest(double from, double to, double
    ↪  expected)
    {
        var actual = Sum.Solve(Accumulate.TailRecursiveInvoke, Term, from,
        ↪  Next, to);
        Assert.Equal(expected, actual);
    }

    [Theory]
    [InlineData(1, 1, 1)]
    [InlineData(1, 2, 3)]
    [InlineData(-5, 5, 0)]
    [InlineData(1, 10, 55)]
```

```
    public void ImperativeInvokeTest(double from, double to, double expected)
    {
        var actual = Sum.Solve(Accumulate.ImperativeInvoke, Term, from, Next,
        ↪  to);
        Assert.Equal(expected, actual);
    }

    [Theory]
    [InlineData(1, 1, 1)]
    [InlineData(1, 2, 3)]
    [InlineData(-5, 5, 0)]
    [InlineData(1, 10, 55)]
    public void DotNetInvokeTest(double from, double to, double expected)
    {
        var actual = Sum.Solve(Accumulate.DotNetInvoke, Term, from, Next, to);
        Assert.Equal(expected, actual);
    }
}
```

Листинг 6: SumUnitTest.cs - тестирование суммы ряда.

```
using System;
using Library;
using Xunit;

namespace Tests;

public class PiSeriesUnitTest
{
    private const double Epsilon = 0.001;

    [Fact]
    public void RecursiveInvokeTest()
    {
        var actual = PiSeries.Solve(Accumulate.RecursiveInvoke);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
    }

    [Fact]
    public void TailRecursiveInvokeTest()
    {
        var actual = PiSeries.Solve(Accumulate.TailRecursiveInvoke);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
    }

    [Fact]
    public void ImperativeInvokeTest()
    {
        var actual = PiSeries.Solve(Accumulate.ImperativeInvoke);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
```

```
    }

    [Fact]
    public void DotNetInvokeTest()
    {
        var actual = PiSeries.Solve(Accumulate.DotNetInvoke);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
    }
}
```

Листинг 7: `PiSeriesUnitTest.cs` - тестирование вычисления приближенного значения числа π.

```
using System;
using Library;
using Xunit;

namespace Tests;

public class SqrtUnitTest
{
    [Theory]
    [InlineData(1, 1)]
    [InlineData(4, 2)]
    [InlineData(8, 2.828427)]
    [InlineData(16, 4)]
    [InlineData(132172371237, 363555.1832074465)]
    [InlineData(int.MaxValue, 46340.95000105199)]
    public void RecursiveInvokeTest(double x, double expected)
    {
        var actual = Sqrt.RecursiveInvoke(x);
        Assert.True(Math.Abs(expected - actual) < Sqrt.Epsilon);
    }

    [Theory]
    [InlineData(1, 1)]
    [InlineData(4, 2)]
    [InlineData(8, 2.828427)]
    [InlineData(16, 4)]
    [InlineData(132172371237, 363555.1832074465)]
    [InlineData(int.MaxValue, 46340.95000105199)]
    public void TailRecursiveInvokeTest(double x, double expected)
    {
        var actual = Sqrt.TailRecursiveInvoke(x);
        Assert.True(Math.Abs(expected - actual) < Sqrt.Epsilon);
    }

    [Theory]
    [InlineData(1, 1)]
    [InlineData(4, 2)]
```

```
    [InlineData(8, 2.828427)]
    [InlineData(16, 4)]
    [InlineData(132172371237, 363555.1832074465)]
    [InlineData(int.MaxValue, 46340.95000105199)]
    public void FixedPointInvokeTest(double x, double expected)
    {
        var actual = EquationSolver.FixedPointOfTransform(y => x / y,
        ↪  EquationSolver.AverageDampTransform, 1.0);
        Assert.True(Math.Abs(expected - actual) < Sqrt.Epsilon);
    }

    [Theory]
    [InlineData(1, 1)]
    [InlineData(4, 2)]
    [InlineData(8, 2.828427)]
    [InlineData(16, 4)]
    [InlineData(132172371237, 363555.1832074465)]
    [InlineData(int.MaxValue, 46340.95000105199)]
    public void NewtonsMethodInvokeTest(double x, double expected)
    {
        var actual =
            EquationSolver.FixedPointOfTransform(y => Math.Pow(y, 2) - x,
            ↪  EquationSolver.NewtonsTransform, 1.0);
        Assert.True(Math.Abs(expected - actual) < Sqrt.Epsilon);
    }
}
```

Листинг 8: `SqrtUnitTest.cs` - тестирование вычисления квадратного корня с помощью метода Герона и метода нахождения неподвижной точки.

```
using System;
using Library;
using Xunit;

namespace Tests;

public class EquationSolverUnitTest
{
    private const double Epsilon = 0.001;

    [Fact]
    public void HalfIntervalSinPiTest()
    {
        var actual = EquationSolver.HalfIntervalMethod(Math.Sin, 2.0, 4.0);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
    }

    [Fact]
    public void HalfIntervalEquationTest()
    {
```

```
        static double Equation(double x)
        {
            return Math.Pow(x, 3) - 2 * x - 3;
        }

        var actual = EquationSolver.HalfIntervalMethod(Equation, 1.0, 2.0);
        Assert.True(Math.Abs(1.8933 - actual) < Epsilon);
    }

    [Fact]
    public void NewtonsSinEquationTest()
    {
        var actual = EquationSolver.FixedPointOfTransform(Math.Sin,
        ↪  EquationSolver.NewtonsTransform, 3.0);
        Assert.True(Math.Abs(Math.PI - actual) < Epsilon);
    }

    [Fact]
    public void NewtonsEquationTest()
    {
        static double Equation(double x)
        {
            return Math.Pow(x, 3) - 2 * x - 3;
        }

        var actual = EquationSolver.FixedPointOfTransform(Equation,
        ↪  EquationSolver.NewtonsTransform, 1.0);
        Assert.True(Math.Abs(1.8933 - actual) < Epsilon);
    }
}
```

Листинг 9: `EquationSolverUnitTest.cs` - тестирование нахождения корней уравнения с помощью метода Ньютона и метода бисекций.

Все юнит-тесты успешно проходят тестирование.

6 Характеристики ПК

Все тесты были произведены на ПК с такими характеристиками:

- ЦПУ - `Intel(R) Core(TM) i5-8265U CPU`:
    1. Базовая скорость: 1.8 ГГц
    2. Количество ядер: 4
    3. Кэш первого уровня: 256 Кб
    4. Кэш второго уровня: 1 Мб
    5. Кэш третьего уровня: 6 Мб
- ОЗУ:
    1. Объём: 16 Гб
    2. Скорость: 2400 МГц
- ЗУ - `Samsung SSD 970 EVO Plus`
    1. Объём: 250 Гб
    2. Среднее время отклика: 1.1 мс

Заключение

В данной работе было показано, как использовать принципы и концепции функционального программирования на примере построение системы вычисления числовых рядов.

Из кода понятно, что громоздкие типы `Func<...,...>` являются нечитабельными, и языку `C#` требуется выведение типов. Этого можно добиться тем, чтобы в язык были добавлены шаблоны, или некие макросы, которые добавили бы элементы метапрограммирования, когда на этапе компиляции мы могли выводить типы, а то и вычислять функции. Только представьте всю мощность функционального программирования и метапрограммирования, когда мы можем уже на этапе компиляции сказать, что данная функция не может быть вычислена из-за ошибки программиста. Естественно не каждую рекурсивную функцию можно вычислить с помощью метапрограммирования бесплатно. Плата за константное вычисление любой рекурсивной функции, запрограммированой с помощью шаблонов является долгая компиляция.

В данной работе было показано, как реализовать простейшую оптимизацию хвостовой рекурсии, которую можно было бы добавить в этап оптимизации кода компилятором. Но разработчики `C#` думают, что необходимо использовать не рекурсию, а пользоваться циклами.

Перечень использованных источников

[1] Gerald J. Abelson, Harold; Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

[2] John McCarthy. History of lisp. *Association for Computing Machinery*, 1978.

[3] Herbert Alexander Simon. *Models of My Life*. The MIT Press, 1996.