Распределённая сборка мусора во время компиляции

Введение в предметную область

Любой системе, где существует выделение динамической памяти, необходим механизм её освобождения.

Распределённая сборка мусора - форма автоматического управления памятью, присутствующая в распределённой системе, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами.

* Актуальность

В распределённых вычислениях, особенно в высокопроизводительных системах, из-за сложности написания программ для суперкомпьютеров, актуальна автоматизация программирования, следовательно и автоматическая сборка мусора. Особенную актуальность автоматическая сборка мусора имеет в системах, где ручное управление памятью не закрывает нужд программистов и, в частности, там требуется решение проблемы сборки мусора.

*LuNA

Система фрагментированного программирования LuNA, разработанная с целью упростить написание параллельных программ, является подходящей системой для исследования проблемы распределённой сборки мусора из-за особенностей архитектуры, позволяющей сосредоточиться на разработке самих алгоритмов.

Цель работы

Целью работы была поставлена разработка и реализация алгоритма сборки мусора для задач численного моделирования и итерационных процессов в системе фрагментированного программирования LuNA.

* Подходы к решению. Динамический сборщик мусора.

- Взвешенный подсчёт ссылок
- Косвенный подсчёт ссылок
- Список ссылок
- Stub-Scion Pair Chains
- Метод трассировки

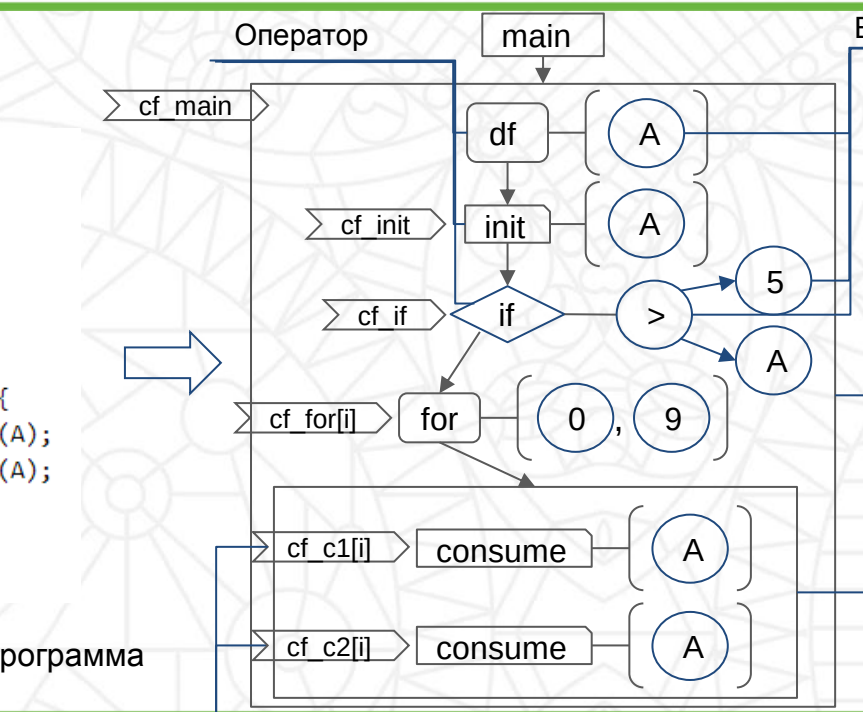
* Подходы к решению. Статический анализ кода.

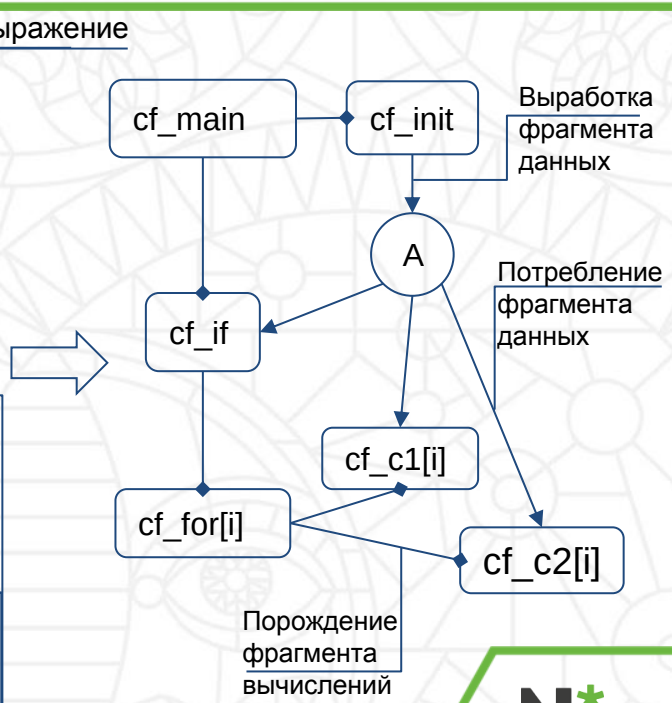
- Граф зависимостей
  - Выделить зависимые узлы
  - Выделить их логически зависимые состояния
  - Представить в удобном для решения задачи виде
- Subscript-by-subscript
- Алгоритм на основе разделов

* Постановка задачи

```
1   sub main()
2   {
3       df A;
4       init(A);
5
6       if (A > 5)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      }
13  }
```

Фрагментированная программа





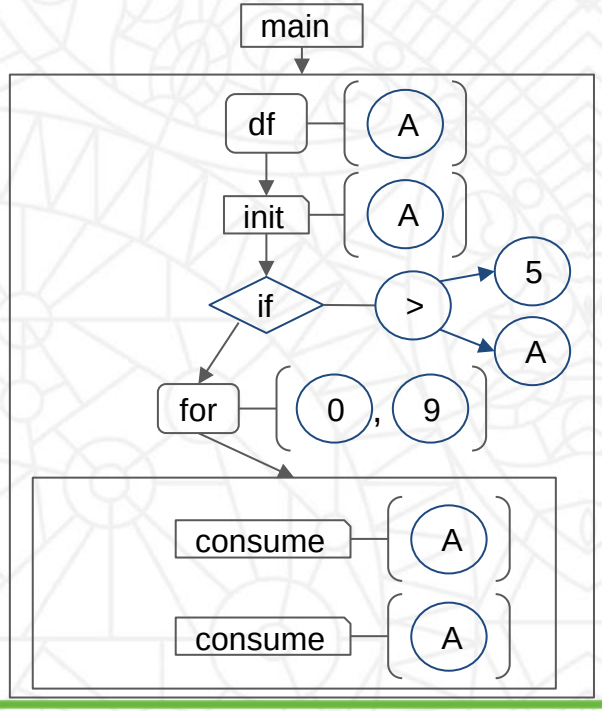
Фрагмент вычислений

AST

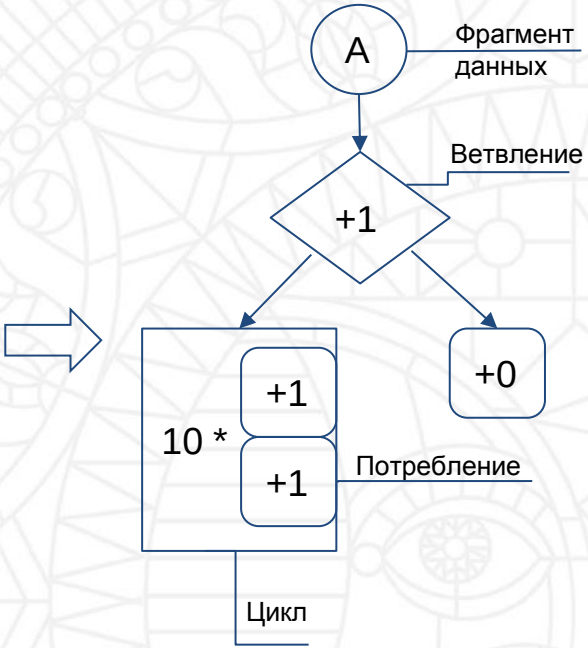
Тело

Множество фрагментов данных и вычислений

Основная идея алгоритма

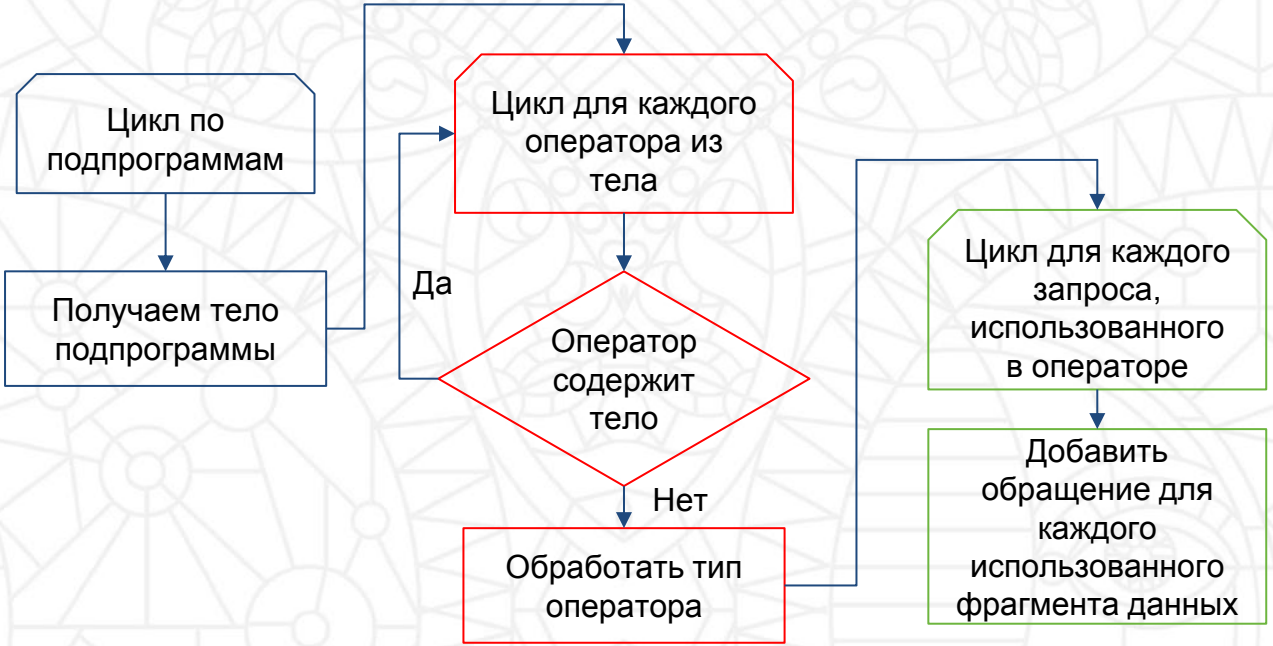


AST



Дерево потреблений

Алгоритм


Цикл по подпрограммам
Получаем тело подпрограммы
Цикл для каждого оператора из тела
Да
Оператор содержит тело
Нет
Обработать тип оператора
Цикл для каждого запроса, использованного в операторе
Добавить обращение для каждого использованного фрагмента данных

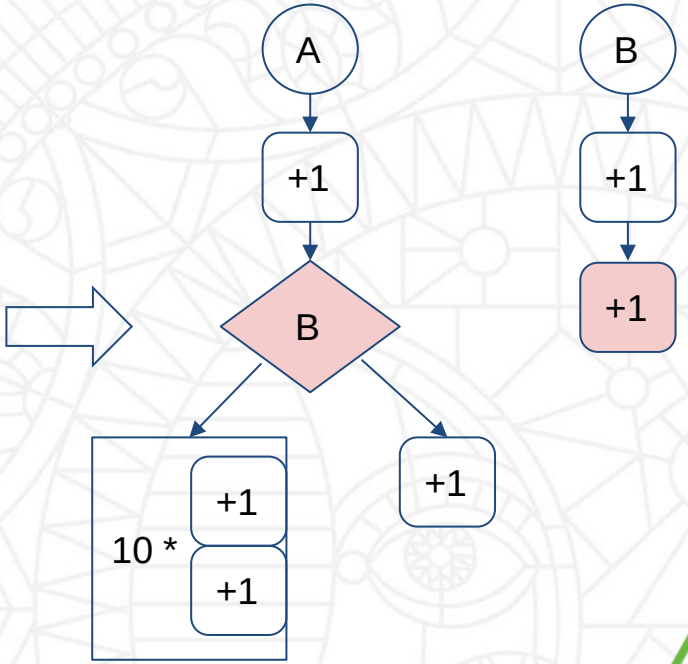
* Описание алгоритма. Рассмотренные проблемы.

- Косвенные потребления
  - Необходимо запросить значение для вычисления числа потреблений фрагмента данных
- Массивы
  - Каждый элемент массива является отдельным фрагментом данных
- Множественные использования в одном выражении или вызове атомарной процедуры
  - Не ведут к повторным обращениям, что особенно актуально для массивов
- Вызов подпрограмм
  - Учитываются потребления фрагментов данных, совершённые в теле подпрограммы
- Зависимость от значения локального фрагмента данных
  - Не приводит к построению неправильного дерева потреблений

Описание алгоритма. Косвенные потребления.

Будем называть потребления косвенными, если они необходимы для вычисления количества потребления других фрагментов данных.

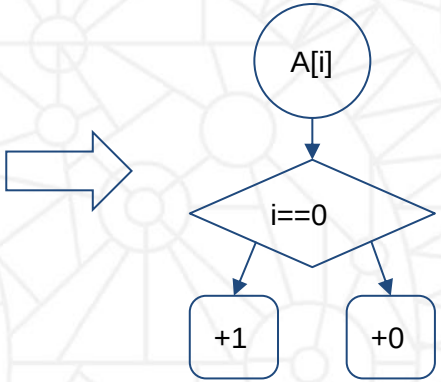
```
1   sub main()
2   {
3       df A, B;
4       consume(A);
5
6       if (B > 5)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      } else {
13          consume(A);
14      }
15  }
```



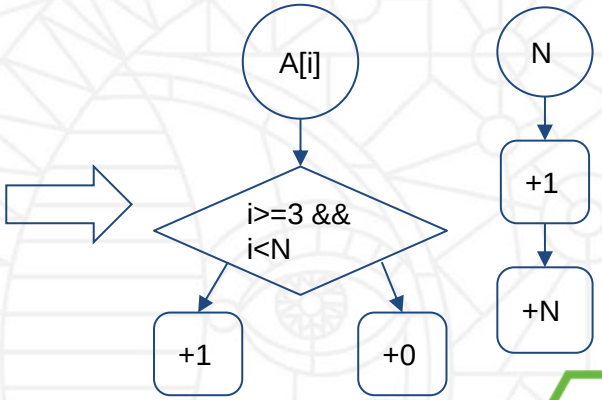
Описание алгоритма. Массивы.

Каждое обращение происходит по условно неизвестному индексу, тем самым потребление добавляется в дерево с некоторым условием.

```
1  sub main()
2  {
3      df A;
4      consume(A[0]);
5  }
```



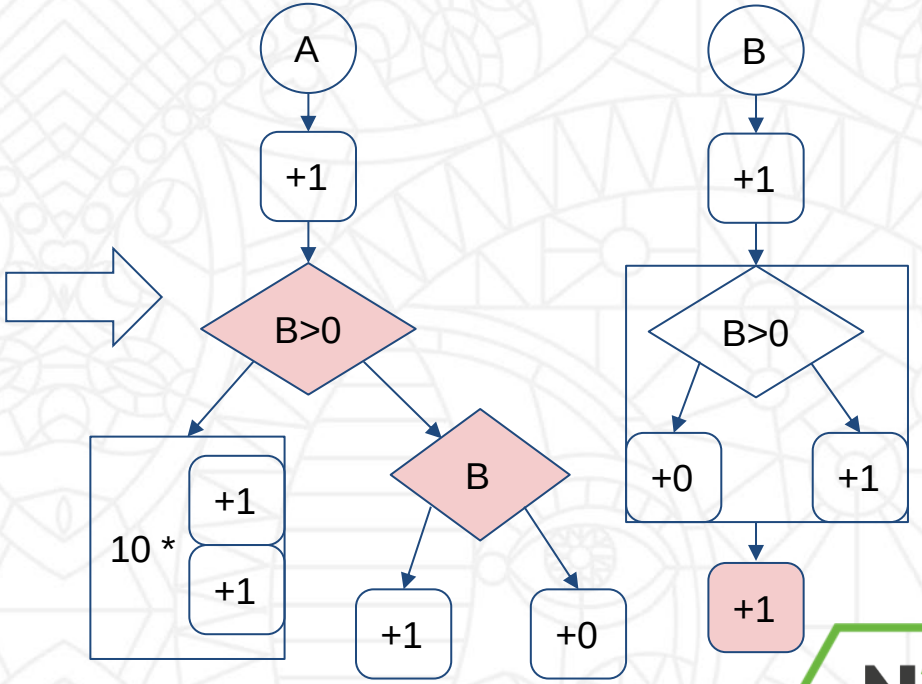

```
1  sub main()
2  {
3      df A, N;
4
5      for i=3..N-1 {
6          consume(A[i]);
7      }
8  }
```



* Описание алгоритма. Множественные использования.

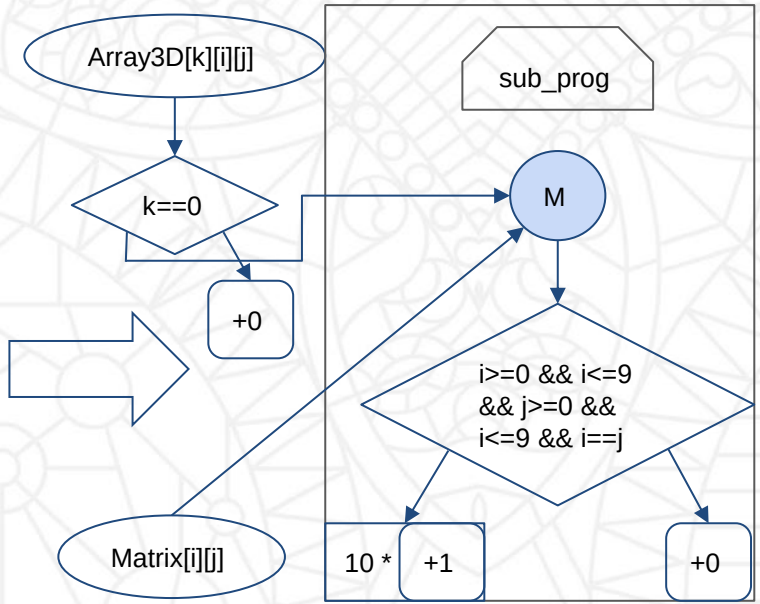
Для каждого дерева создаётся агрегатор, не позволяющий, собирающий все обращения к одному и тому же фрагменту данных в одно.

```
1   sub main()
2   {
3       df A, B;
4       consume(A);
5
6       if (B > 0)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      } else {
13          if (B == 0)
14          {
15              consume(A);
16          }
17      }
18  }
```



Описание алгоритма. Вызов подпрограммы.

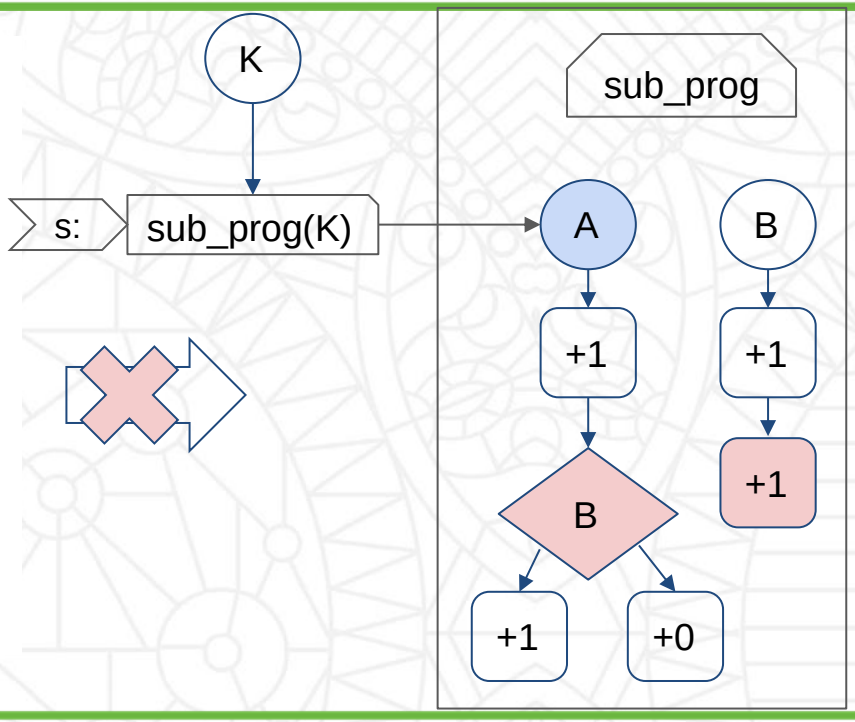
```
1   sub matrix(name M)
2   {
3       for i=0..9 {
4           consume(M[i][i]);
5       }
6   }
7
8   sub main()
9   {
10      df Array3D, Matrix;
11
12      matrix(Array3D[0]);
13      matrix(Matrix);
14  }
```



1. Построение множества деревьев потреблений для каждой подпрограммы
2. Сопоставление аргументов подпрограммы и переданных значений

* Описание алгоритма. Зависимость с разницей в области видимости.

```
1   sub sub_prog(name A)
2   {
3       df B;
4       init(1, B);
5       consume(A);
6
7       if (B > 0)  {
8           consume(A);
9       }
10  }
11
12  sub main()
13  {
14      df K;
15      cf s: sub_prog(K);
16  }
```



Способы решения проблемы:

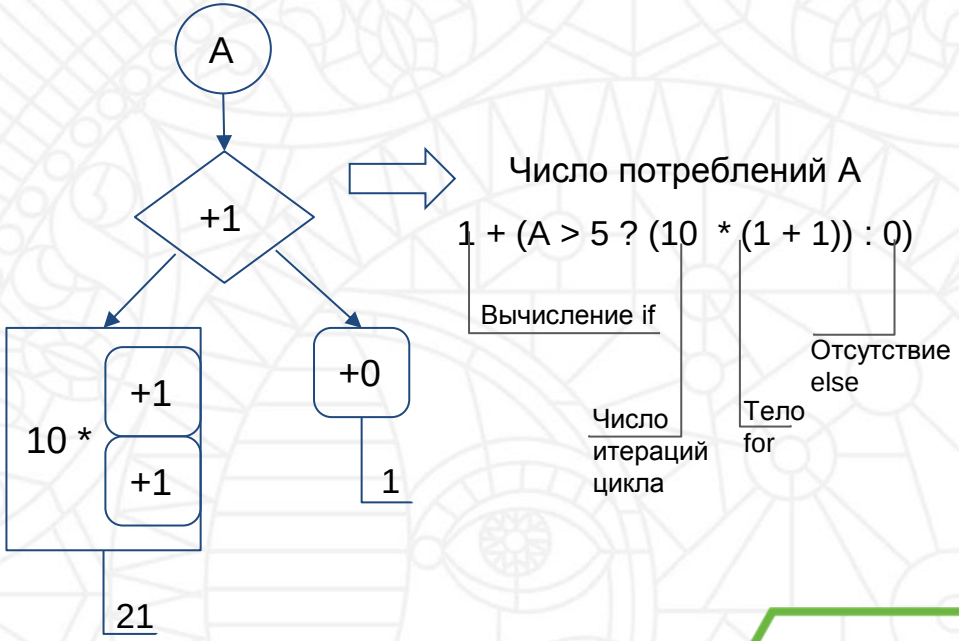
1. Глобальные идентификаторы фрагментов данных, доступные на этапе компиляции
2. Рекомендация “Присваивание”

* Описание алгоритма. Интерпретация результата.

Во время компиляции мы не можем:

- определить какая именно ветвь будет выполнена
- освободить память, которая ещё не была выделена

Вставка дерева в исходный код программы позволит системе исполнения определить, какая из ветвей будет выполнена, и предоставит информацию о моменте, подходящем для освобождения памяти.

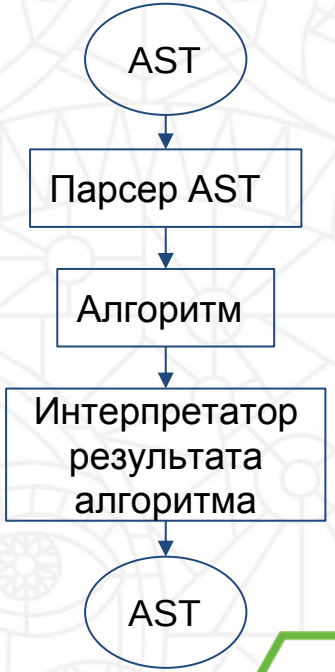


Реализация

Алгоритм реализован в виде модуля для компилятора на языке программирования Python.

Ключевая часть алгоритма, ответственная за хранение зависимостей между фрагментами данных реализована в виде независимого набора классов.

Алгоритм использует рекомендации – механизм ручного управления поведением системы исполнения, позволяющий сообщить компилятору или системе исполнения дополнительную информацию

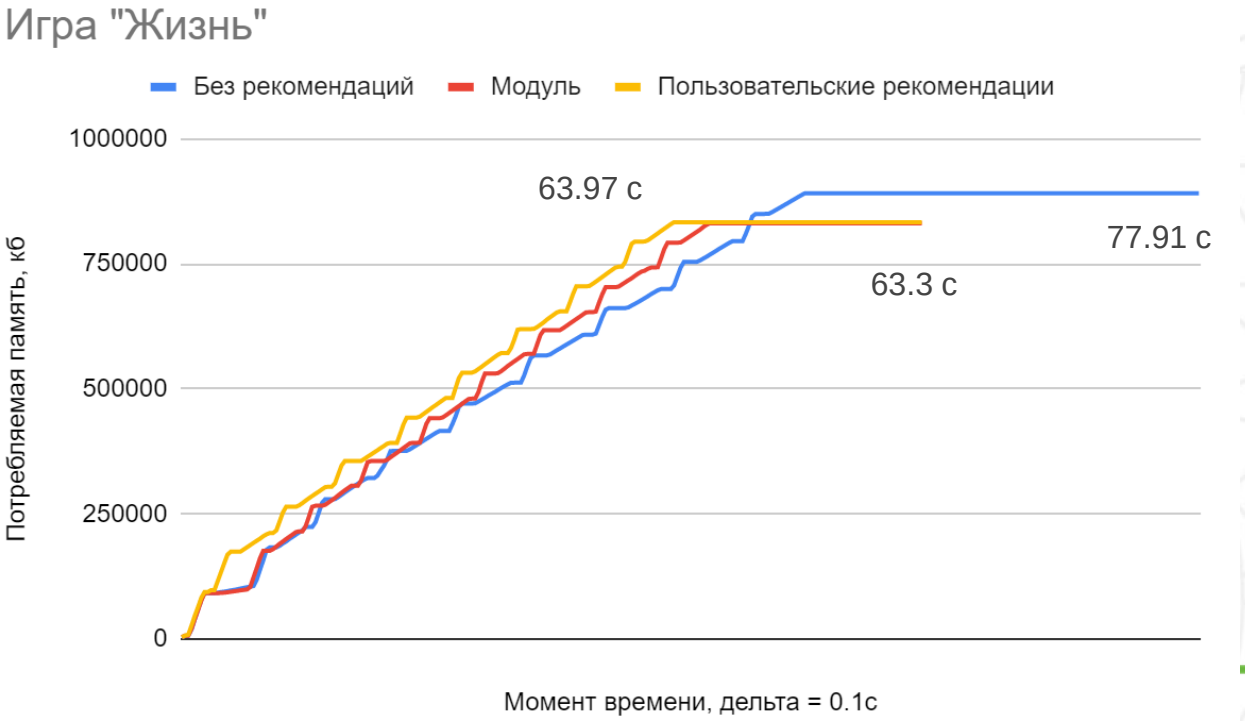


Описание задачи. Игра “Жизнь”.

Варианты запусков:

- без рекомендаций
  - не освобождает память
- пользовательские рекомендации
  - освобождаются красные клетки
- модуль для компилятора
  - освобождается всё поле

Результаты. Игра “Жизнь”.

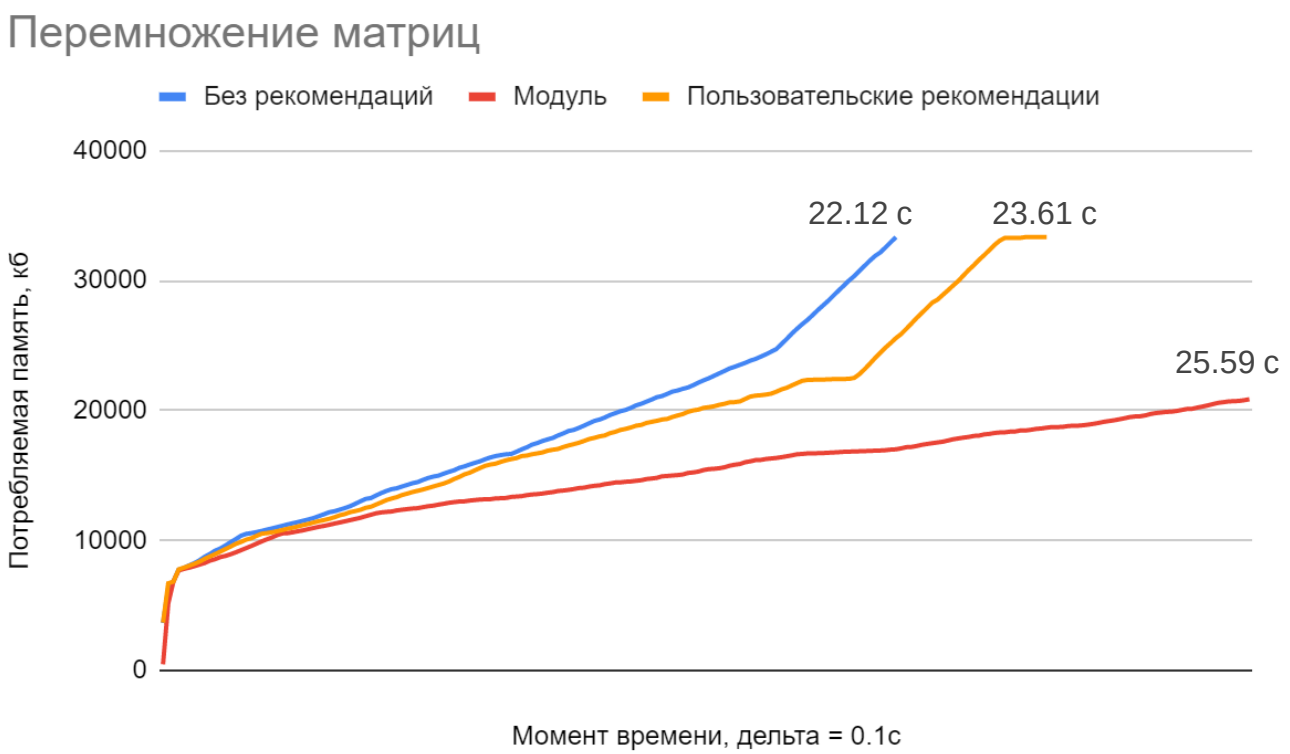


Описание задачи. Перемножение матриц.

Варианты запусков:

- без рекомендаций
  - не освобождает память
- пользовательские рекомендации
  - освобождаются матрицы, помеченные красным
- модуль для компилятора
  - освобождаются все матрицы

Результаты. Перемножение матриц.



* Практическая значимость

Практическая значимость решения заключается в создании сборщика мусора в распределённой системе в качестве автоматического механизма управления памятью вместо использования механизмов ручного управления, что позволит:

- снизить порог вхождения в систему для новых пользователей
- ускорить процесс написания программ, при этом не теряя эффективности по памяти
- снизить количество ошибок, связанных с ручным управлением памятью, таких как преждевременное или несвоевременное удаление неиспользуемых объектов

Планы на будущее

В дальнейшем планируется продолжить работу над сборкой мусора в системе фрагментированного программирования LuNA. Возможными направлениями для развития являются:

1. Расширение множества поддерживаемых операторов и конструкций языка.
2. Реализация глобальных идентификаторов фрагментов данных, доступных во время компиляции.
3. Разработка динамического алгоритма в качестве альтернативного механизма сборки мусора.

Заключение

Был разработан и реализован алгоритм автоматической сборки мусора, производящий анализ зависимостей, определение числа использований фрагментов данных и последующую модификацию программы для своевременного освобождения уже неиспользуемой памяти.

- Разработан алгоритм автоматического анализа зависимостей, определения числа использований фрагментов данных и последующей модификации программы для своевременного освобождения уже неиспользуемой памяти.
- Разработанный алгоритм был реализован и интегрирован в систему фрагментированного программирования LuNA.
- Проведено экспериментальное исследование, которое показало работоспособность разработанного алгоритма.