Лекция 5. Управление динамической памятью и сборка мусора

Коновалов А. В.

8 мая 2024 г.

Управление динамической памятью

Управление динамической памятью

На предшествующих занятиях мы рассмотрели реализацию только двух классов памяти программ: статическая и автоматическая (стековая) память.

- Статическая память выделяется под глобальные переменные на этапе компиляции, время жизни переменных — всё время выполнения программы.
- Автоматическая память выделяется под локальные переменные. Время жизни локальных переменных соответствует времени выполнения конкретного вызова функции. Память выделяется по принципу LIFO.

Нерассмотренной остаётся динамическая память. **Динамические переменные** выделяются по запросу программиста, время их жизни не привязано к времени существования стекового кадра.

Ручное и автоматическое управление динамической памятью

На предыдущем слайде ничего не сказано про завершение времени жизни динамических переменных.

- В случае ручного управления памятью время жизни определяется программистом: когда динамическая переменная становится не нужна, программист выполняет команду её освобождения (функция `free()` в Си, оператор `delete` в C++).
- В случае автоматического управления памятью временем жизни управляет сам язык программирования: память освобождается, когда среда времени выполнения решает её освободить. Здесь используются механизмы вроде подсчёта ссылок, сборки мусора, управления памятью на основе регионов и т.д.

Организация кучи

Организация кучи

Область памяти, отводимая под динамические переменные, носит название **кучи**.

Существует много подходов к организации кучи, мы рассмотрим следующие:

- выделять память инкрементом указателя,
- связный список блоков,
- алгоритм близнецов (buddy algorithm),
- пулы объектов разного размера (подход библиотеки `jemalloc` и языка Go).

Инкремент указателя (1)

Это самый простой метод. Глобальный указатель содержит адрес начала свободной области, при каждой аллокации он инкрементируется на требуемый размер:

```
(var heap 100500)
(var free ptr "=" heap)

(function alloc (size)
  (var res ptr)
  (res "=" (L free))
  (free "=" ((L free) "+" (L size)))
  (return (L res))
)
```

Инкремент указателя (2)

Преимуществами метода являются эффективность и простота.

Недостаток — невозможность освобождать динамическую память в произвольном порядке. Можно освобождать либо в порядке LIFO, либо можно освободить всю память за раз (`(free "=" heap)`).

Рассматривается этот метод по нескольким причинам:

- В некоторых случаях какая-то группа переменных в динамической памяти освобождается одновременно. Например, при завершении одного из проходов компиляции структуры данных, использовавшиеся в нём, становятся ненужными.
- Этот подход в организации кучи является основой нескольких алгоритмов сборки мусора: уплотняющий сборщик мусора, копирующий сборщик Чени и сборщик мусора по поколениям.

Связный список блоков (1)

Память, выделенная в куче, организована в виде блоков переменной длины, объединённых в двунаправленный список. Блоки в списке упорядочены по возрастанию адресов. Каждый из блоков может быть помечен либо свободным, либо занятым.

Операция выделения памяти пробегает по списку блоков и ищет свободный блок требуемого размера. Если размер найденного блока больше требуемого, то он разбивается на два: блок требуемого размера и свободный блок остатка, первый помечается как занятый. Иначе блок просто помечается как занятый. Адрес блока возвращается пользователю.

Функция освобождения памяти принимает адрес блока и помечает его свободным, а затем проверяет соседей. Если среди соседей есть свободные блоки, то они сливаются.

Связный список блоков (2)

Преимуществами алгоритма является простота реализации.

Недостатки:

- быстродействие — поиск свободного блока может занять продолжительное время,
- фрагментация памяти — свободная память разбита на много мелких блоков, запрос памяти большого объёма может завершиться неудачей, даже если суммарный объём свободных блоков более чем достаточен.

Различают две стратегии: первый подходящий и наилучший подходящий.

Стратегия «наилучший подходящий» снижает фрагментацию, но выделение памяти будет выполняться медленнее.

Связный список блоков (3) — реализация

```
(struct Block
  (Block_flags int)
  (Block_size int)
  (Block_prev ptr)
)

(const USED "=" 1)
(const FREE "=" 0)
(const HEAD_SIZE = 100500)

(var heap HEAP_SIZE)
```

```
(function init_heap ()
  (var last ptr)
 ((heap "+" Block_flags) "=" FREE)
  ((heap "+" Block_size) "="
   (HEAP_SIZE "-" (2 "*" Block)))
  ((heap "+" prev) "=" 0)

  (last "="
    (HEAP_SIZE "-" Block))
 (((L last) "+" Block_flags) "=" USED)
 (((L last) "+" Block_size) "=" 0)
 (((L last) "+" Block_prev) "=" heap)
)
```

Связный список блоков (4) — реализация

```
(function malloc (size)
  (var (b ptr))
  (b "=" heap)
  (while ((L ((L b) "+" Block_size)) "<>" 0)
    (if (((L ((L b) "+" Block_flags)) "==" FREE)
         and ((L ((L b) "+" Block_size)) "⩾" (L size)))
      (call split_block (L b) (L size))
      (((L b) + Block_flags) "=" USED)
      (return ((L b) + Block))
    else
      (b "=" (L ((L b) "+" (Block_size "+" Block))))
    )
  )
  (return 0)
)
```

Используется стратегия «первый подходящий».

Связный список блоков (5) — реализация

```
(function split_block (b size)
  (var (nextb ptr)
  (if ((L ((L b) "+" Block_size)) ">" ((L size) + Block))
    (nextb "=" ((L b) "+" (Block "+" size))
    (((L nextb) "+" Block_flags) "=" FREE)
    (((L nextb) "+" Block_size) "="
      ((L ((L b) "+" Block_size)) "-" ((L size) "+" Block))
    )
    (((L nextb) "+" Block_prev) "=" (L b))

    (((L b) "+" Block_size) "=" (L size))
  )
)
```

Алгоритм близнецов (buddy algorithm) (1)

Идея алгоритма:

- Размер кучи для данного алгоритма должен быть степенью двойки.
- Размер выделяемого блока памяти также является степенью двойки.
- Зная адрес блока, можно за константное время найти адрес его «близнеца» — соседнего блока, который образовался в результате разбиения родительского блока: смещение относительно начала кучи близнеца равно исключающему ИЛИ (XOR) между смещением кучи конкретного блока и его размером.

Алгоритм близнецов (2)

Идея алгоритма (продолжение):

- Свободные блоки объединяются в двунаправленные списки блоков равного размера.
- Поддерживается массив $FREE$, где $FREE[i]$ элемент содержит указатель на список блоков размером 2^{i+m}, где 2^m — минимальный размер блока (например, 4 слова).
- По факту, блоки хранят служебную информацию (размер, занят/свободен), поэтому доступный пользователю размер будет равен $2^k - s$, где s — размер служебной информации.
- Изначально все списки $FREE[i]$ пустые, кроме последнего, последний хранит единственный блок, соответствующий всей куче целиком.

Алгоритм близнецов (3)

Алгоритм выделения памяти размером $size$:

- находится такое минимальное r, что $size \leq 2^{r+m} - s$;
- если для всех $n \geq r$ списки $FREE[n]$ пусты, значит, выделить память невозможно;
- пока список $FREE[r]$ пуст, повторять:
  - найти минимальное n, такое что $FREE[n]$ не пусто;
  - удалить блок из $FREE[n]$, разбив его на два близнеца и поместить их в список $FREE[n-1]$;
- пометить первый блок из $FREE[r]$ как занятый и вернуть его пользователю.

Алгоритм близнецов (4)

Алгоритм освобождения блока:

- Найти близнеца блока. Близнец может быть занят, свободен или разбит на меньшие блоки.
- Если близнец свободен, исключить близнеца из соответствующего списка $FREE[i]$ и слить освобождаемый блок с близнецом. Повторить алгоритм для объединённого блока.
- Пометить блок свободным и добавить в список $FREE[i]$ для соответствующего размера.

Для того, чтобы вычислить смещение относительно начала кучи близнеца, нужно выполнить исключающее ИЛИ (XOR) между смещением кучи конкретного блока и его размером.

Алгоритм близнецов (5)

Преимущества алгоритма близнецов:

- Эффективность — и выделение, и освобождение памяти выполняются за время, пропорциональное логарифму от размера кучи.
- Снижение фрагментации — эксперименты демонстрируют низкую фрагментацию памяти этого алгоритма.

Недостатки:

- Перерасход памяти — около половины памяти будет теряться из-за округления вверх до $2^r - s$.
- Размер кучи должен быть степенью двойки.

Пулы объектов (1)

- Куча делится на некоторое количество пулов — массивов блоков различных размеров.
- Размеры блоков в пулах образуют приблизительно экспоненциально возрастающий ряд.
- При выделении памяти требуемый размер округляется до минимального допустимого размера блока и выбирается свободный блок и соответствующего пула. Блок помечается как занятый.
- По адресу динамической переменной можно определить пул, из которого она была выделена, а значит и размер, и смещение в этом пуле.
- Освобождение сводится к поиску соответствующего пула и пометке блока как свободного.
- Пометки блоков могут осуществляться в битовых картах, соответствующих пулам, т.е. размер служебной информации составляет 1 бит.
- Свободные блоки удобно объединить в однонаправленный список — первое слово указывает на следующий свободный блок в пуле.

Пулы объектов (2)

Пример. Размеры пулов в менеджере памяти Go:

```
const _NumSizeClasses = 67
var class_to_size = [_NumSizeClasses]uint16{0, 8, 16
, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192,
208, 224, 240, 256, 288, 320, 352, 384, 416, 448,
480, 512, 576, 640, 704, 768, 896, 1024, 1152, 1280,
 1408, 1536, 1792, 2048, 2304, 2688, 3072, 3200,
3456, 4096, 4864, 5376, 6144, 6528, 6784, 6912, 8192
, 9472, 9728, 10240, 10880, 12288, 13568, 14336,
16384, 18432, 19072, 20480, 21760, 24576, 27264,
28672, 32768}
```

Источник: https://habr.com/ru/companies/ruvds/articles/442648/

Пулы объектов (4)

Преимущества:

- Высокое быстродействие.
- Можно эффективно идентифицировать блок выделенной памяти, зная не только адрес начала блока, но и адрес, находящийся в середине блока (в Go без этого не напишешь сборщик мусора).

Недостатки:

- Фрагментация памяти — память, доступная в пулах для меньших размеров, недоступна для выделения памяти большего размера.

Сборка мусора

Сборка мусора (1)

Сборка мусора — это эмуляция компьютера с бесконечной памятью (Реймонд Чен).[1]

Строго говоря, есть разные подходы в эмуляции компьютера с бесконечной памятью (см. диссертацию Рафаэля Пруста (Raphaël L. Proust) «ASAP: As Static As Possible memory management» (2017)[2]).

Задачей является определение множества динамических переменных, которые больше не потребуются выполняемой программе. Определение этого множества алгоритмически неразрешимо, поэтому на практике его аппроксимируют.

---

[1] Источник: https://devblogs.microsoft.com/oldnewthing/20100809-00/?p=13203, перевод на русский: https://www.transl-gunsmoker.ru/2012/09/blog-post.html

[2] https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-908.pdf

Сборка мусора (2)

Сборка мусора аппроксимирует множество ненужных программе динамических переменных множеством переменных, недостижимых из **корневого множества** — множества ссылок на переменные, доступных в программе непосредственно.

Корневое множество обычно включает глобальные переменные, переменные в стеке вызовов и в регистрах процессора.

Динамическая переменная считается достижимой, если на неё ссылается ссылка из корневого множества, либо из другой достижимой переменной. Переменная может быть достижимой, но при этом в программе неиспользуемой.

Сборка мусора (3) — пример достижимой неиспользуемой переменной

```
class Stack {
  private Object[] items = new Object[100];
  private int top = 0;

  public void push(Object value) {
    items[top++] = value;
  }

  public Object pop() {
    return items[--top];
  }
}
```

Где ошибка?

Сборка мусора (4) — пример достижимой неиспользуемой переменной

```
class Stack {
  private Object[] items = new Object[100];
  private int top = 0;

  public void push(Object value) {
    items[top++] = value;
  }

  // ошибка исправлена
  public Object pop() {
    Object value = items[--top];
    items[top] = null;
    return value;
  }
}
```

Расширения НУИЯПа для сборки мусора (1)

Для удобства дальнейшего обсуждения потребуются формальные обозначения для ряда понятий и операций. Введём их в НУИЯП:

```
t.Definition ::= … | t.GlobalRefs | t.DynVarType
t.GlobalRefs ::= (refs s.Name*)
t.DynVarType ::=
  (dynvar s.Name
    (refs s.Name*)?
    (s.Name t.ConstExpr)*
  )
t.LocalVars ::= (var (refs s.Name*)? (s.Name t.ConstExpr)*)
t.Statement ::= …
  | (t.Expr ":-" t.Expr)
  | (gc-alloc t.Expr s.Name)
  | (ref-return t.Expr)
```

Расширения НУИЯПа для сборки мусора (2)

Ссылки — переменные размера 1 слово. Они намеренно выделены, т.к. должны учитываться сборщиком мусора. Значением ссылки может быть либо адрес объекта в куче, либо `0`. Все ссылки при создании неявно инициализирутся нулями.

Глобальные ссылки — имена глобальных переменных, хранящих ссылки.

`dynvar` определяет тип динамической переменной. Значения этого типа могут создаваться *только* в куче.

Блок локальных переменных расширяется определением переменных-ссылок.

Расширения НУИЯПа для сборки мусора (3)

Присваивание ссылок синтаксически отличается от присваивания слов и записывается знаком `":-"` (обозначение позаимствовано из Симулы-67). Левым операндом должен быть адрес ссылки — переменной-ссылкой (глобальной или локальной) либо поля `dynvar`'а, описанное в секции `(refs …)`.

(Забегая вперёд, отметим, различать два присваивания потребуется в алгоритме подсчёта ссылок.)

Инициализатор `(gc-alloc t.Expr s.Name)` получает адрес ссылки и имя типа динамической переменной. Он создаёт в куче новый объект и присваивает его ссылке.

Оператор `(ref-return t.Expr)` используется, если возвращаемое значение — ссылка.

Расширения НУИЯПа для сборки мусора (4)

Будем называть переменные-ссылки и поля-ссылки динамических переменных **управляемыми переменными**.

Отметим, что объекты кучи «в безопасности» только, когда их адреса хранятся в управляемой переменной. Если адрес сохранить в неуправляемой переменной или неявно на стеке как часть подвыражения, то сборщик мусора их не увидит и может освободить память — ссылка станет висячей.

```
/* опасно! */
(call sum_array (call make_array (L N)))

/* безопасно */
(array ":-" (call make_array (L N)))
(call sum_array (L array))
```

Расширения НУИЯПа для сборки мусора (5)

Присваивать ссылке возвращаемое значение можно, только если функция завершается `ref-return`'ом:

```
(function f (…)
  …
  (ref-return …)
)

(function g (…)
  …
  (r ":-" (call f …))
  …
)
```

Алгоритмы сборки мусора

Мы рассмотрим несколько алгоритмов сборки мусора:

- подсчёт ссылок,
- пометить и подмести (mark-and-sweep),
- уплотняющий сборщик мусора,
- копирующий сборщик мусора Чени,
- сборка мусора по поколениям,
- консервативная сборка мусора.

Далее мы будем использовать слово **объект** как синоним **динамической переменной**.

Общим для всех алгоритмов сборки мусора является то, что в каждый объект неявно включается дополнительная информация для сборщика: информация о типе объекта, флаги, счётчики и т.д.

Подсчёт ссылок (1)

Относить ли этот алгоритм к категории «сборка мусора» — спорный вопрос. «Но нам не шашечки, нам ехать»☺.

С каждым объектом ассоциируется счётчик, который показывает, сколько ссылок в памяти программы содержат его адрес.

При создании объекта счётчик ссылок устанавливается в 1:

```
(gc-alloc ‹перем› ‹тип›)
```

Во время выполнения программы счётчики ссылок могут как инкрементироваться, так и декрементироваться. Если после декремента счётчик достигает нуля, объект уничтожается — декрементируются счётчики ссылок объектов, на которые он ссылается и память, занятая им, освобождается.

Подсчёт ссылок (2)

При выполнении присваивания:

`(‹перем› ":-" ‹адрес›)`

счётчик ссылок для `‹адреса›` увеличивается на 1, счётчик ссылок для объекта, хранившегося в `‹перем›`, на 1 уменьшается. Порядок операций важен, т.к. иначе в случае `(r ":-" (L r))` можно получить утечку памяти.

Подсчёт ссылок (3)

Присваивание возвращаемого значения обрабатывается особым образом:

`(‹перем› ":-" (call ‹функция› ‹аргументы›))`

Оператор `(ref-return ‹адрес›)` инкрементирует счётчик у объекта, поэтому при присваивании счётчик ссылок у правой части инкрементировать не надо.

При завершении функции (в эпилоге) счётчики ссылок для всех переменных-ссылок декрементируются.

Подсчёт ссылок (4)

Преимущества алгоритма:

- простота реализации,
- возможность библиотечной реализации в языках без сборки мусора: C++, Rust,
- отсутствие пауз в работе (хотя удаление одного объекта может привести к каскаду удалений других объектов).

Недостатки алгоритма:

- невозможность работы с циклическими зависимостями.

Пометить и подмести (mark and sweep)

Алгоритм выполняется в несколько фаз:

- Пометка всех объектов в куче, как недостижимых.
- Перебор всех ссылок в корневом множестве (глобальные и локальные ref-переменные):
  - из каждой ссылки инициируем обход графа объектов (как правило, в глубину) с пометкой всех достижимых.
- Освобождение памяти, занятой объектами, помеченными как недостижимые (фаза подметания).

Преимущество алгоритма:

- простота реалиализации.

Недостаток алгоритма:

- не предотвращает фрагментацию памяти.

Уплотняющий сборщик мусора (1)

Этот сборщик мусора также обходит кучу, помечая в ней достижимые объекты, однако вместо фазы подметания выполняется фаза уплотнения.

Фаза уплотнения состоит из следующих проходов:

1. Проход вычисления новых адресов для достижимых объектов — для каждого объекта вычисляется и запоминается его новый адрес.
2. Обход всех ссылок в корневом множестве и внутри достижимых объектов с коррекцией адресов.
3. Перемещение объектов на новые адреса.

В данном алгоритме свободная память всегда образует непрерывную область.

Обозначим `loc[o]` — ассоциативный массив, отображающий адреса достижимых объектов на новые адреса.

Уплотняющий сборщик мусора (2)

Фазу уплотнения можно описать следующим псевдокодом:

```
free = 0
for o in ‹перечисляем объекты в порядке возрастания адресов›:
    if o.достижим:
        loc[o] = free
        free += sizeof(*o)

for r in ‹ссылки из корневого множества и достижимых объектов›:
    *r = loc[*r]

free = 0
for o in ‹перечисляем объекты в порядке возрастания адресов›:
    if o.достижим:
        memmove(free, o, sizeof(*o))
        free += sizeof(*o)
```

Уплотняющий сборщик мусора (3)

Преимущество алгоритма:

- предотвращает фрагментацию памяти — создание объетов выполняется инкрементом указателя.

Недостатки:

- сложность алгоритма,
- необходимость перемещения объектов в памяти.

Копирующий сборщик мусора Чени (1)

Данный сборщик мусора в каждый момент времени использует только половину доступной памяти под выделение объектов.

Куча делится на две полукучи, объекты выделяются в одной из них, вторая остаётся свободной.

Когда половина заполняется, все достижимые объекты копируются в другую половину и полукучи меняются ролями.

Копирующий сборщик мусора Чени (2)

В алгоритме используется вспомогательная процедура `NewLoc(o)`, принимающая адрес объекта в «старой» полукуче и возвращающая адрес в «новой».

Обозначим

- `loc[o]` — ассоциативный массив, отображающий адреса объектов из «старой» кучи, в «новую».
- `free` — указатель на начало свободной области памяти в «новой» куче.
- `unscanned` — указатель на непросканированный объект в «новой» куче.

Изначально `free = unscanned = ‹начало новой кучи›`.

Копирующий сборщик мусора Чени (3)

Алгоритм может быть описан следующим псевдокодом:

```
def NewLoc(o):
    if o not in loc:
        memcpy(free, o, sizeof(*o))
        loc[o] = free
        free += sizeof(*o)
    return loc[o]

free = unscanned = ‹начало новой кучи›

for r in ‹ссылки из корневого множества›:
    *r = NewLoc(*r)

while unscanned < free:
    for r in ‹ссылки из объекта *unscanned›:
        *r = NewLoc(*r)
    unscanned += sizeof(*unscanned)
```

Копирующий сборщик мусора Чени (4)

Заметим, что алгоритм нерекурсивный и обходит граф объектов в ширину.

Преимущества алгоритма:

- отсутствие фрагментации памяти,
- повышение локальности данных,
- простая и эффективная реализация,
- алгоритм анализирует только достижимые объекты в куче (используется в Chicken Scheme самым грязным образом☺).

Недостатки алгоритма:

- Половина памяти не используется под выделение объектов.

Сборка мусора по поколениям (1)

Исследования показывают, что большинство объектов «умирают» молодыми, т.е. становятся мусором в течение небольшого количества времени (выполненных инструкций) после их выделения.

(Например, в Java для конструирования строки может быть создан объект `StringBuilder`, который живёт всего один фрейм стека.)

В результате куча будет заполняться мусором из короткоживущих объектов.

Сборка мусора по поколениям (2)

Сборка мусора по поколениям предполагает разбиение памяти на несколько куч, т.н. поколений.

Новые объекты всегда выделяются в самом младшем поколении («нулевом поколении», «эдеме»).

Сборка мусора в i-м поколении напоминает алгоритм Чени: достижимые объекты из j-го поколения ($j \leq i$) копируются в $(i+1)$-е.

Сборка мусора в самом последнем поколении выполняется уже каким-то другим алгоритмом (например, mark-and-sweep или уплотняющий сборщик).

Сборка мусора по поколениям (3)

С точки зрения сборки мусора в i-м поколении, ссылки на объекты из старших поколений являются корневым множеством.

Очевидно, обходить всю кучу в поисках таких ссылок неэффективно (размер «эдема» составляет мегабайты, размер всей кучи может достигать гигабайтов). Поэтому рассматривается т.н. **запомненное множество** — множетство ссылок из старших поколений, содержащих адреса из младших поколений.

Соответственно, в фазе сборки мусора анализируются не вся память старших поколений, а только ссылки из запомненного множества.

Сборка мусора по поколениям (4)

Запомненное множество обновляется в операторе присваивания:

(‹цель› ":-" ‹адрес›)

Если ‹цель› находится внутри объекта поколения s, а ‹адрес› — внутри поколения j, причём $j < s$, то адрес ‹цели› нужно добавить в запомненное множество.

Запоминать можно либо непосредственно адрес ссылки, либо адрес объекта, содержащего ссылку, либо вообще страницу в памяти, содержащую ссылку.

Запоминание может осуществляться вызовом специальной процедуры при выполнении присваивания, либо даже средствами защиты памяти самой ОС.

Сборка мусора по поколениям (5)

Преимущества алгоритма:

- снижение фрагментации памяти,
- повышение локальности данных (эдем может почти целиком влезать в кэш),
- уменьшение пауз в работе.

Недостаток алгоритма:

- сложность.

Консервативная сборка мусора (1)

Рассмотренные алгоритмы предполагают, что для каждого слова в памяти можно определить, является ли оно управляемой ссылкой.

Этого можно добиться, соответствующим образом разметив память: во фреймах стека и в самих объектах должна храниться служебная информация о том, по каким смещениям располагаются адреса.

Однако, такое не возможно в небезопасных языках (например, в Си), либо нецелесообразно из каких-то других соображений (Go).

Консервативная сборка мусора выполняется в предположении, что любое слово в памяти может хранить ссылку на объект. Если содержимое слова выглядит как адрес выделенного объекта, то соответствующий объект считается достижимым.

Консервативная сборка мусора (2)

Преимущество алгоритма:

- возможность использовать сборку мусора в небезопасных языках, либо упрощение среды поддержки времени выполнения («рантайма») в безопасных языках,
- возможность хранить указатели на середину объектов (Go).

Недостатки алгоритма:

- переоценка множества достижимых объектов,
- невозможность реализации перемещающих или уплотняющих алгоритмов сборки мусора; фактически, остаётся доступен только алгоритм «пометить и подмести» с оптимизациями вроде частичной сборки мусора (в курсе рассматриваться не будут).

Реализация сборки мусора

Общие соображения

Расширения НУИЯПа, как и при рассмотрении ООП, в некоторых случаях окажутся синтаксическим сахаром (т.е. выражаются через конструкции базового НУИЯПа), в некоторых случаях потребуют особой, «нативной» реализации (как вызовы методов в ООП).

И также, как и при рассмотрении ООП, будем резервировать идентификаторы с двумя прочерками подряд «__» для нужд реализации.

Многие расширения НУИЯПа компилируются похожим образом для разных алгоритомов сборки мусора. Поэтому мы сначала рассмотрим общую схему, а затем частности для разных алгоритмов.

Расширение модельного ассемблера

Модельный ассемблер поддерживает встроенную константу `PROGRAM_SIZE`, которая содержит размер загруженной программы. Можно считать, что `:PROGRAM_SIZE` неявно дописывается ассемблером в самый конец исходника.

В библиотеку поддержки НУИЯПа добавляется определение

```
:_START_OF_HEAP__
PROGRAM_SIZE
```

которое определяет переменную `START_OF_HEAP__`, содержащую адрес начала кучи.

(В лекции 3 определена переменная `PROGRAM_SIZE` с тем же значением, можно пользоваться ею.)

Инициализация кучи

Будем считать, что размер кучи определяется как 90 % от доступного объёма свободной памяти.

Свободная память начинается с адреса `START_OF_HEAP__` и завершается текущим значением стека вызовов. Адрес стека вызовов можно примерно оценить при помощи следующей функции:

```
(function approx_getsp ()
  (vars (local 1))
  (return local)
)
```

Соответственно, на стек вызовов будет отводиться 10 % доступной памяти.

Компиляция глобальных ссылок

Имена ссылок, перечисленных во всех (`refs e.Names`), присутствующих в программе, объединяются в общий список и компилируются в список переменных, инициализированных нулями и предварённых переменной `global_refs__`, содержащей их количество.

```
(refs One Two Three)

(function f (x y) …)

(var (n 10) (m 100))

(refs Four Five)
```

```
(var global_refs__ 5)
(var One 1 "=" 0)
(var Two 1 "=" 0)
(var Three 1 "=" 0)
(var Four 1 "=" 0)
(var Five 1 "=" 0)

(function f (x y) …)

(var (n 10) (m 100))
```

Компиляция типов динамических переменных (1)

`dynvar` компилируется в структуру, которая начинается с блока служебной информации типа `gc_info__` (её содержимое зависит от алгоритма). Управляемые ссылки компилируются в поля размера 1 после `gc_info__`.

Для каждого типа определяется константа `NREFS_‹имя›__`, содержащая количество объектных ссылок в структуре.

Важно, что все поля-ссылки расположены по соседству в начале объекта (сразу после блока служебной информации), поэтому для обхода ссылок сборщику мусора достаточно знать их количество. Если бы язык допускал бы чередования управляемых ссылок и прочих данных, то потребовалось бы хранить битовую карту или список смещений ссылок в структуре.

Компиляция типов динамических переменных (2)

Компиляция выглядит так:

```
(dynvar ‹имя›
  (refs ‹ссылка₁› ‹ссылка₂›…)
  (‹поле₁› ‹размер₁›)
  (‹поле₂› ‹размер₂›)
  …
)
```

```
(struct ‹имя›
  ("-" gc_info__)
  (‹ссылка₁› 1)
  (‹ссылка₂› 1)
  …
  (‹поле₁› ‹размер₁›)
  (‹поле₂› ‹размер₂›)
  …
)

(const NREFS_‹имя›__ "=" …)
```

Компиляция локальных переменных-ссылок (1)

```
t.LocalVars ::= (var (refs s.Name*)? (s.Name t.ConstExpr)*)
```

Базовый НУИЯП для локальных переменных резервирует участок стека ниже старого значения `FP`.

НУИЯП со сборкой мусора незначительно меняет эту схему.

- Поверх старого значения `FP` (и, соответственно, со смещением -1 относительно базы) сохраняется количество управляемых ссылок (имён в секции `refs`).
- Затем резервируется память на управляемые ссылки, причём в прологе они должны быть проинициализированы нулями.
- Затем резервируется память под обычные локальные переменные — она никак не инициализируется.

Если управляемых ссылок среди локальных переменных нет (секция `refs` пустая или отсутствует), то под адресом базы располагается число `0`.

Компиляция локальных переменных-ссылок (2)

Благодаря такой организации фреймов, можно перечислить все управляемые ссылки, находящиеся на стеке:

- Функция (`call getFP`) позволяет получить адрес базы в текущем фрейме.
- Фреймы провязаны в однонаправленный список: адрес базы каждого фрейма ссылается на адрес базы предыдущего фрейма (последний у функции `main` содержит `0`).
- По смещению `-1` от базового адреса фрейма хранится количество управляемых ссылок.
- По смещениям `-2` и далее находятся сами объектные ссылки.

Компиляция новых операторов

Реализация операторов `":-"`, `gc-alloc` и `ref-return` существенно зависит от используемого алгоритма.

Однако, в большинстве алгоритмов `":-"` будет эквивалентен обычному оператору присваивания `"="`, а `ref-return` — обычному `return`.

В дальнейшем мы будем их упоминать, только если они имеют какую-то нестандартную реализацию.

Компиляция счётчиков ссылок (1)

Алгоритм будет использовать функции `alloc__`, `add_ref__`, `release__`, `assign__` и `assign_call__`, написанные на базовом НУИЯПе.

- `(call alloc__ size nrefs)` — создаёт динамическую переменную размером `size` и содержащую `nrefs` управляемых ссылок.
- `(call add_ref__ ptr)` — увеличивает на 1 счётчик ссылок, связанный с объектом, возвращает свой аргумент.
- `(call release__ ptr)` — декрементирует счётчик ссылок для объекта. Если счётик доходит до нуля, объект уничтожается.
- `(call assign__ target object)` — присваивает объект переменной по данному адресу.
- `(call assign_call__ target object)` — специальный случай присваивания.

Компиляция счётчиков ссылок (2)

Блок служебной информации имеет вид:

```
(struct gc_info__
  ‹поля реализации кучи›
  (gc_info_nrefs__ int)
  (gc_info_rc__ int)
)
```

- `‹поля реализации кучи›` — служебные поля, зависящие от реализации кучи. Например, для кучи, организованной как двусвязный список блоков здесь будет записан размер объекта, поле флагов и ссылка на предыдущий блок.
- Поле `gc_info_nrefs__` содержит количество управляемых ссылок в объекте.
- Поле `gc_info_rc__` — счётчик ссылок на объект.

Компиляция счётчиков ссылок (3)

Операторы компилируются в вызов соответствующих функций:

`(‹цель› ":-" (call …))`	`(call assign_call__ ‹цель› (call …))`
`(‹цель› ":-" ‹адрес›)`	`(call assign__ ‹цель› ‹адрес›)`
`(ref-return ‹выраж›)`	`(return (call add_ref__ ‹выраж›))`
`(gc-alloc ‹перем› ‹тип›)`	`(call assign_call__ ‹перем›` `(call alloc__ ‹тип› NREFS_‹тип›__))`

Компиляция счётчиков ссылок (4)

Реализация некоторых функций:

```
(function add_ref__ (obj)
  (if ((L obj) "==" 0) (return 0))
  (((L obj) "+" gc_info_rc__) "="
    ((L ((L obj) "+" gc_info_rc__)) "+" 1))
  (return (L obj))
)

(function assign__(target obj)
  (call add_ref__ (L obj))
  (call release__ (L (L target)))
  ((L target) "=" (L obj))
)

(function assign_call__(target obj)
  (call release__ (L (L target)))
  ((L target) "=" (L obj))
)
```

Компиляция счётчиков ссылок (5)

Реализация некоторых функций:

```
(function release__ (obj)
  (var (p ptr) (end ptr))
  (if ((L obj) "==" 0) (return 0))
  (((L obj) "+" gc_info_rc__) "="
    ((L ((L obj) "+" gc_info_rc__)) "-" 1))
  (if ((L ((L obj) "+" gc_info_rc__)) "==" 0)
    (p "=" ((L obj) "+" gc_info__))
    (end "=" (p "+" (L ((L obj) "+" gc_info_nrefs__))))
    (while ((L p) "<" (L end))
      (call release__ (L p))
      (p "=" ((L p) "+" 1))
    )
    ‹код освобождения блока ptr, зависит от реализации кучи›
  )
)
```

Компиляция счётчиков ссылок (6)

В эпилоге функции неявно добавляется код, вызывающий `release__` для каждой локальной переменной, являющейся управляющей ссылкой.

Количество управляемых ссылок, лежащее поверх старого значения `FP`, в данной реализации будет не нужно.

Компиляция «пометить и подмести» (1)

Блок служебной информации имеет вид:

```
(struct gc_info__                (const FREE__ 0)
  ‹поля реализации кучи›         (const UNREACHED__ 1)
  (gc_info_flags__ int)          (const UNSCANNED__ 2)
  (gc_info_nrefs__ int)          (const SCANNED__ 3)
)
```

- `‹поля реализации кучи›` — служебные поля, зависящие от реализации кучи. Например, для кучи, организованной как двусвязный список блоков здесь будет записан размер объекта и ссылка на предыдущий блок.
- Поле `gc_info_flags__` — поле флагов, *совместно* используется и реализацией кучи, и сборщиком мусора.
- Поле `gc_info_nrefs__` содержит количество управляемых ссылок в объекте.

Компиляция «пометить и подмести» (2)

`":-"` и `ref-return` компилируются в `"="` и `return`.

Оператор выделения памяти

`(gc-alloc ‹перем› ‹тип›)`

компилируется в

`(‹перем› "=" (call alloc__ ‹тип› NREFS_‹тип›__))`

Функция `alloc__(size nrefs)` пытается выделить память под объект требуемого размера, если не удаётся, то вызывает сборку мусора.

Компиляция «пометить и подмести» (3)

Важно иметь ввиду, что сборка мусора выполняется в условиях недостатка доступной памяти, а глубина стека вызовов ограничена. Поэтому наивная рекурсивная реализация обхода графа объектов в глубину неприемлема.

Память, требуемую для выполнения обхода, можно резервировать в каждом объекте. Например, нерекурсивная реализация обхода в глубину подразумевает использование явного стека. Стек может быть организован как однонаправленный список объектов, который может быть представлен дополнительным полем `gc_info_next__` в служебном блоке в начале объекта.

Другой интересный вариант описан в книге

- Вирт Н., Гуткнехт Ю. Разработка операционной системы и компилятора. Проект Оберон: Пер. с англ. Борисов Е.В., Чернышов Л.Н. — М.: ДМК Пресс, 2012. — 560 с.: ил.

Компиляция уплотняющего сборщика мусора

Свободное место в куче всегда представлено непрерывным куском, поэтому выделение памяти под новые объекты выполняется простым инкрементом указателя.

Блок служебной информации имеет вид:

```
(struct gc_info__
  (gc_info_size__ int)
  (gc_info_flags__ int)
  (gc_info_nrefs__ int)
  (gc_info_new_addr__ ptr)
)
```

Смысл первых трёх полей очевиден. Последнее поле `gc_info_new_addr__` используется в роли ассоциативного массива `loc[o]` — содержит новый адрес объекта после уплотнения.

Компиляция сборщика мусора Чени (1)

Свободное место в куче всегда представлено непрерывным куском, поэтому выделение памяти под новые объекты выполняется простым инкрементом указателя.

Блок служебной информации имеет вид:

```
(struct gc_info__
  (gc_info_size__ int)
  (gc_info_nrefs__ int)
  (gc_info_new_addr__ ptr)
)
```

Если поле `gc_info_new_addr__` хранит нулевое значение, значит объект ещё не был скопирован в новую кучу и процедура `NewLoc(o)` должна будет его скопировать.

Если значение `gc_info_new_addr__` ненулевое, значит объект уже скопирован, а в старой полукуче осталась от него лишь пустая оболочка.

Компиляция сборщика мусора по поколениям (1)

Мы рассмотрим простую реализацию с двумя поколениями: эдемом и большой кучей.

Эдем выделяется как массив сравнительно небольшого размера:

```
(const EDEN_SIZE__ "=" …)
(var eden__ EDEN_SIZE__)
```

Запомненное множество представляется хэш-множество в виде таблицы с прямой адресацией:

```
(const EDEN_REFS_CAPACITY__ "=" …)
(var eden_refs_size__ int "=" 0)
(var eden_refs__ EDEN_REFS_CAPACITY__)
```

Компиляция сборщика мусора по поколениям (2)

Оператор присваивания

`(‹цель› ":-" ‹адрес›)`

компилируется в вызов вспомогательной функции

`(call assign__ ‹цель› ‹адрес›)`

которая, если `‹цель›` — адрес в большой куче, а `‹адрес›` — адрес объекта в эдеме, запоминает `‹цель›` в хэш-множестве.

Компиляция сборщика мусора по поколениям (3)

Малая сборка мусора выполняется, когда либо эдем, либо запомненное хэш-множество переполнились (так что даже присваивание может инициировать сборку мусора).

Большая сборка мусора выполняется, когда свободное место в основной куче оказалось меньше эдема (а значит, следующая малая сборка мусора может не завершиться успешно).

Реализация консервативной сборки мусора (1)

Консервативная сборка мусора не требует дополнительной разметки, поэтому ref-поля и ref-переменные компилируются как обычные поля и обычные переменные. Соответственно, количество ref-полей в динамических переменных и количество локальных ref-переменных на стеке нигде сохранять и подсчитывать не надо.

Реализации `":-"` и `ref-return`, очевидно, будут идентичны `"="` и `return` соответственно.

В сущности, консервативную сборку мусора можно добавить и в базовый, и в объектно-ориентированный НУИЯП.

Реализация консервативной сборки мусора (2)

Оператор создания объекта

```
(gc-alloc ‹перем› ‹имя›)
```

компилируется в

```
(‹перем› "=" (call alloc__ ‹имя›))
```

т.к. количество ссылок внутри объекта для сборщика мусора не важно.

Реализация консервативной сборки мусора (3)

Блок служебной информации имеет вид:

```
(struct gc_info__
  ‹поля реализации кучи›
  (gc_info_flags__ int)
)

(const FREE__ 0)
(const UNREACHED__ 1)
(const UNSCANNED__ 2)
(const SCANNED__ 3)
```

- `‹поля реализации кучи›` — служебные поля, зависящие от реализации кучи.
- Поле `gc_info_flags__` — поле флагов, *совместно* используется и реализацией кучи, и сборщиком мусора.

Реализация консервативной сборки мусора (4)

В качестве корневого множества используются участки адресного пространства от адреса `0` до `START_OF_HEAP__` (там находятся глобальные переменные, хоть и в перемешку с кодом) и весь стек вызовов.

Нижнюю границу стека можно получить, взяв адрес локальной переменной.

Верхнюю границу стека нужно запоминать в ассемблерном коде до вызова функции `main`. В лекции 3 мы её сохраняем в переменной `MEMORY_SIZE`.

Реализация консервативной сборки мусора (5)

Основная сложность алгоритма — определение того, является ли содержимое слова адресом выделенного объекта.

- При использовании списка блоков адреса выделенных блоков нужно помещать в множество, представленное деревом (например, АВЛ) или хэш-таблицей. Использование дерева обеспечит сложность $O(\log N)$, однако позволит работать с указателями на середину объектов.
- При использовании близнецового алгоритма можно по данному указателю (вернее, по смещению от начала кучи) за $O(\log N)$ операций найти блок памяти, которому он принадлежит.
- При использовании пулов объектов можно найти пул за время $O(p)$ или $O(\log p)$, где p — количество пулов. В самом пуле искомый объект находится за константу (как частное от смещения относительно начала пула и размера элемента).

Другие аспекты сборки мусора: массивы, слабые ссылки и финализаторы

Реализация массивов (1)

Язык Java позволяет создавать либо массивы, которые не содержат объектных ссылок (массивы примитивных типов), либо массивы, которые состоят только из объектных ссылок (массивы типа класс или массивы типа интерфейс). Это здорово упрощает реализцию сборки мусора.

Сделаем как в Java. Расширим НУИЯП следующими операторами:

```
t.Statement ::= …
  | (gc-alloc-val-array t.Address t.Expr)
  | (gc-alloc-ref-array t.Address t.Expr)
```

Первый оператор распределяет массив размера `t.Expr` слов, которые не являются управляемыми ссылками. Второй оператор распределяет массив `t.Expr` управляемых ссылок.

Реализация массивов (2)

Реализация массивов в нашей модели очевидна.

- Оператор `gc-alloc-val-array` выделяет динамическую переменную указанного размера, в поле `gc_info_nrefs__` записывает `0`.
- Оператор `gc-alloc-ref-array` в поле `gc_info_nrefs__` записывает размер массива и всё его содержимое заполняет нулями.

Разумеется, в консервативной реализации эти два оператора являются синонимами.

Реализация массивов (3)

Для доступа к элементам массива потребуется двуместный оператор

```
t.Expr ::= …
  | (t.Expr "@" t.Expr)
```

Первый аргумент — ссылка на массив, второй аргумент — индекс. Возвращаемое значение — адрес ячейки памяти с данным индексом:

```
(var (refs xs))
(gc-alloc-val-array xs 3)
((xs "@" 1) "=" 7)
((xs "@" 2) "=" (L (xs "@" 1)))
```

Слабые ссылки (1)

Сильная ссылка — ссылка, которая владеет объектом. Если объект был доступен через цепочку сильных ссылок до сборки мусора, то он остаётся существовать и после сборки мусора.

Ссылки, определяемые ключевым словом `refs` (глобальные, локальные, поля) — сильные ссылки.

Слабая ссылка — ссылка, которая не владеет объектом. Если объект был доступен *только* через слабые ссылки, то после сборки мусора он уничтожается, а слабые ссылки обнуляются.

Во многих реализациях языков программирования слабые ссылки доступны как библиотечные классы с именами вроде `WeakRef` или `WeakPtr`.

Примеры: `java.lang.ref.WeakReference` (Java), `System.WeakReference` (C#), `std::weak_ptr` (C++), модуль `weakref` в Python, `std::rc::Weak` (Rust).

Слабые ссылки (2)

Расширим наш язык операторами для работы со слабыми ссылками:

```
t.Statement ::= …
  | (gc-alloc-weak-ref t.Expr)
  | (gc-weak-ref-read t.Expr t.Expr)
  | (gc-weak-ref-write t.Expr t.Expr)
```

Первый оператор создаёт объект слабой ссылки, второй и третий позволяют обращаться к его содержимому.

Слабые ссылки (3)

Объект слабой ссылки можно рассматривать как объект с единственным ссылочным полем, операторы чтения и записи можно рассматривать как методы доступа к нему.

- Оператор (`gc-alloc-weak-ref ‹перем›`) принимает адрес переменной и присваивает ей объект слабой ссылки.
- Оператор (`gc-weak-ref-read ‹перем› ‹wref›`) присваивает сильной ссылке `‹перем›` содержимое объекта слабой ссылки `‹wref›`.
- Оператор (`gc-weak-ref-write ‹wref› ‹объект›`) присваивает слабой ссылке `‹wref›` ссылку на `‹объект›`.

Слабые ссылки (4) — подсчёт ссылок

Сборщик мусора подсчётом ссылок теперь должен содержать два счётчика — число сильных и число слабых ссылок:

```
(struct gc_info__
  ‹поля реализации кучи›
  (gc_info_nrefs__ int)
  (gc_info_rc__ int)
  (gc_info_wc__ int)
)
```

Счётчик `gc_info_wc__` отслеживает количество слабых ссылок, которые указывают на этот объект. Он может меняться при выполнении оператора `gc-weak-ref-write` по аналогии с оператором присваивания (у правого операнда число ссылок увеличивается, у левого — уменьшается).

Слабые ссылки (5) — подсчёт ссылок

Объект *разрушается* (т.е. вызывается `release__` для всех его ссылочных полей), когда его счётчик сильных ссылок `gc_info_rc__` обнуляется.

Памать, выделенная объекту, *освобождается*, когда оба счётчика становятся равными нулю.

Если объект слабой ссылки ссылается на объект с `gc_info_rc__`, равным нулю, то оператор чтения `gc-weak-ref-read` для него будет возвращать нулевой указатель.

Слабые ссылки (6) — подсчёт ссылок

При уничтожении объекта слабой ссылки счётчик слабых ссылок для объекта декрементируется. Если после этого оба счётчика оказались равными нулю, выделенная память освобождается.

Уничтожение объекта выполняется функцией рантайма `release__`. Поэтому важно, чтобы эта функция могла отличать объект слабой ссылки от других объектов.

Для этого можно полю `gc_info_nrefs__` присваивать отрицательное значение.

Дополнение. Интересные особенности реализации слабых ссылок в Swift: https://habr.com/ru/articles/341014/.

Слабые ссылки (7) — пометить и подмести

Объекты слабых ссылок можно представить как следующие объекты:

```
(dynvar weak_ref__
  (weak_ref_value__ ptr)
  (weak_ref_next__ ptr)
)

(var weak_refs__ ptr "=" 0)
```

- Поле `weak_ref_value__` хранит адрес объекта, на который ссылка ссылается.
- Слабые ссылки провязаны в однонаправленный список (поле `weak_ref_next__`), на который указывает `weak_refs__`.

Заметим, что все ссылки выше неуправляемые (`ptr`) и `weak_refs__` формально не входит в корневое множество.

Реализация операторов `gc-alloc-weak-ref`, `gc-weak-ref-read` и `gc-weak-ref-write` очевидна.

Слабые ссылки (8) — пометить и подмести

Переменная `weak_refs__` не является управляемой ссылкой, поэтому на стадии пометки она будет проигнорирована. Связный список, на который она ссылается, может содержать как недостижимые слабые ссылки, так и слабые ссылки, ссылающиеся на недостижимые объекты.

После стадии пометки нужно выполнить обход однонаправленного списка слабых ссылок.

- Если ссылка осталась недостижимой — она удаляется из этого списка (память, выделенная ей, будет освобождена на стадии подметания).
- Если ссылка ссылается на объект, помеченный как недостижимый, то полю `weak_ref_value__` следует присвоить нулевой указатель.

В уплотняющем сборщике мусора слабые ссылки реализуются точно также.

Слабые ссылки (9) — сборщик Чени

Слабые ссылки описываются той же структурой `weak_ref__` и тем же списком `weak_refs__`, реализация трёх методов для работы с ними та же.

После выполнения сборки мусора часть звеньев списка `weak_refs__` скопировалась в новую полукучу (в старой остались «остатки» с `gc_info_new_addr__`, указывающие на новые объекты), часть просто осталась в старой куче. Заметим, что адреса `weak_ref_value__` и `weak_ref_next__` не поменялись.

Сборщик мусора опять должен обойти этот список, оставив в нём звенья, которые скопировались в новую полукучу. В этих звеньях он должен скорректировать поля `weak_ref_value__`, установив новые адреса для скопированных объектов и обнулив для не скопированных.

В сборке мусора по поколениям слабые ссылки реализуются аналогично.

Финализаторы (1)

Финализаторы — специальные методы объектов в ОО-языках, которые вызываются перед уничтожением объекта сборщиком мусора.

Расширим НУИЯП следующим оператором:

```
t.Statement ::= …
  | (gc-add-finalizer t.Expr t.Expr)
```

Оператор `(gc-add-finalizer ‹объект› ‹функция›)` связывает с объектом указатель на функцию. Функция должна принимать один параметр — ссылку на объект.

Перед уничтожением объекта эта функция будет вызвана.

Считаем, что поведение при многократном вызове `gc-add-finalizer` для одного и того же объекта, при восстановлении ссылки из функции финализатора (например, присваивании глобальной переменной), вызове `gc-add-finalizer` из финализатора не определено.

Финализаторы (2) — подсчёт ссылок

Добавить финализаторы в управление памятью подсчётом ссылок несложно:

- В структуру `gc_info__` нужно добавить новое поле, которое будет содержать указатель на функцию.
- Функция `release__` при обнулении счётчика ссылок вызывает финализатор (если он установлен), после чего освобождает (`release__`) ссылочные поля объекта и освобождает выделенную память.

Финализаторы (3) — пометить и подмести

Поддерживается список финализаторов:

```
(dynvar finalizer__
  (finalizer_object__ ptr)
  (finalizer_function__ ptr)
  (finalizer_next__ ptr)
)

(var finalizers__ ptr)
```

- Оператор `gc-add-finalizer` добавляет звено в этот список.
- После стадии пометки выполняется обход этого списка.
  - Звенья списка, ссылающиеся на недостижимые объекты, из списка удаляются, для объектов вызываются финализаторы.
  - Звенья списка, ссылающиеся на достижимые объекты, помечаются как достижимые.

Финализаторы (4)

В уплотняющем сборщике мусора реализация аналогична.

В сборщике мусора Чени и сборщике по поколениям звенья списка финализаторов для достижимых объектов нужно копировать в новую полукучу, остальное аналогично.

Предложенный алгоритм небезопасен: финализатор может нарушить инварианты сборщика мусора, например, присвоить финализируемый объект полю достижимого объекта или глобальной переменной.

Безопасность можно повысить, если все объекты, подлежащие финализации, пометить как достижимые (или, соответственно, перенести их в другую полукучу/поколение), собрать мусор, а потом финализировать. Да, они сразу после этого станут мусором, но зато инварианты сохранятся.

Замечание о консервативной сборке

В консервативной сборке слабые ссылки и финализацию мы рассматривать не будем, т.к. подход аналогичен алгоритму «пометить и подмести», но сложностей в реализации больше.