Конструирование компиляторов. Углубленный курс

Николай Кудасов

9 октября 2025

Проект. Часть 1. Сборка мусора

Содержание

2.1. Кратко о проекте

На этом этапе проекта вам необходимо реализовать сборщик мусора для среды времени исполнения языка программирования Stella[1]. Компилятор, среда времени исполнения (включая заглушку для сборщика мусора), а также ряд тестовых программ вам доступны сразу. Ваша задача — заменить заглушку на работающий сборщик мусора:

1. Реализовать один из предложенных алгоритмов сборки мусора.
2. Реализовать сбор и отображение статистики работы сборщика мусора.

2.2. Требования к реализации

Основная цель проекта — реализовать *Сборщик мусора*, часть среды времени исполнения, ответственную за выделение и освобождение памяти рабочей программы.

Реализация проекта допускается на любом языке программирования, по предварительному согласованию с преподавателем. В том числе допускается реализация на языках с поддержкой FFI для языка C, а также генерация кода сборщика мусора (на языке C) при запуске программы на другом языке.

2.2.1. Интерфейс сборщика

Реализация сборщика мусора должна удовлетворять как минимум интерфейсу, описанному в файле `gc.h`:

1. макрос `GC_READ_BARRIER(object, field_index, read_code)` должен быть определён для раскрытия чтения поля объекта; если барьера на чтение нет, достаточно раскрыть макрос в `read_code`;

[1] https://fizruk.github.io/stella/

2. макрос `GC_WRITE_BARRIER(object, field_index, contents, write_code)` должен быть определён для раскрытия записи поля объекта; если барьера на запись нет, достаточно раскрыть макрос в `write_code`;

3. процедура `void* gc_alloc(size_t size_in_bytes)` должна реализовывать механизм выделения блока памяти, размером как минимум `size_in_bytes` байт;

4. процедура `void gc_read_barrier(void *object, int field_index)` должна реализовывать механизм барьера на чтение;

5. процедура `void gc_write_barrier(void *object, int field_index, void *contents)` должна реализовывать механизм барьера на запись;

6. процедура `gc_push_root(void **object)` добавляет ссылку на новый корень программы в множество отслеживаемых корней;

7. процедура `gc_pop_root(void **object)` убирает ссылку на корень программы из множество отслеживаемых корней;

8. процедура `void print_gc_alloc_stats()` печатает статистику использования сборщика мусора на момент вызова;

9. процедура `void print_gc_state()` печатает состояние сборщика мусора на момент вызова;

2.2.2. Алгоритм сборки мусора

Для выполнения проекта достаточно реализовать

1. (на выбор) копирующую сборку мусора (используя semi-DFS подход [1, Algorithm 13.11]) или сборку мусора алгоритмом пометок (используя разворот указателей [1, Algorithm 13.6]);

2. (на выбор) с поддержкой сборки по поколениям [1, §13.4] или инкрементальной сборки [1, §13.5];

Каждый алгоритм может предполагать ряд параметров, которые представлены макросами, передающимися во время компиляции модуля сборки мусора. В этом проекте достаточно реализовать один параметр:

- `MAX_ALLOC_SIZE` — максимальный размер памяти для кучи (на молодое поколение) в байтах.

По согласованию с преподавателем возможна реализация вариаций или других сборщиков мусора (например, по материалам в научных статьях).

2.2.3. Статистика сборщика

Сборщик мусора должен собирать статистику работы и печатать её при вызове процедуры `print_gc_alloc_stats()`. Информация может быть распечатана в произвольном формате, но должна содержать:

1. Общее количество выделенной памяти (в байтах и в блоках/объектах)

2. Общее количество сборок мусора, всего и на каждое поколение

3. Максимальное количество используемой памяти (выделенной, но ещё не собранной) всего и по каждому поколению

4. Количество чтений и записей в памяти (контролируемой сборщиком)

5. Количество срабатываний барьера на чтение/запись

2.2.4. Отладочная информация

Сборщик мусора должен уметь печатать отладочную информацию для инспектирования состояния сборщика в произвольный момент времени, при вызове процедуры `print_gc_state()`. Информация может быть распечатана в произвольном формате, но должна содержать:

1. Состояние кучи:
   (a) Для каждого объекта на куче должны быть указаны адрес объекта, поля объекта, «шапка» или специальные поля выделенного объекта (если применимо)
   (b) Объекты должны быть разделены по поколениям
   (c) Границы областей памяти (для каджого поколения)
   (d) Список/множество свободных блоков (если применимо)
   (e) Состояние внутренних переменных сборщика мусора (например, `scan`, `next`, `limit` при реализации копирующего сборщика мусора)
2. Множество корней программы (как минимум, все указатели на кучу, доступные из корней)
3. Количество выделенной памяти в данный момент (на каждое поколение)
4. Количество свободной памяти (на каждое поколение)

2.2.5. Документация

Проект должен быть документирован

1. развёрнутыми комментариями в исходном коде
2. файлом `README`, в котором обозначены
   (a) инструкции по сборке и запуску проекта
   (b) описаны параметры сборщика мусора (значение, допустимые значения, и т.п.)
   (c) описан формат вывода процедуры `print_gc_alloc_stats()`, включая наглядный пример
   (d) описан формат вывода процедуры `print_gc_state()`, включая наглядный пример

2.3. Компиляция и запуск программ

Программы на языке Stella состоят из одного модуля, содержащимся в одном файле (обычно с расширением `.stella` или `.st`).

2.3.1. Из Stella в C

Для компиляции в C, используйте следующую команду (в примере используется Docker контейнер):

```
docker run -i fizruk/stella compile < ФАЙЛ.stella > ФАЙЛ.c
```

Также можно воспользоваться компилятором в онлайн-песочнице[2] языка Stella:

1. Введите код программы
2. Нажмите кнопку **Compile**
3. Сохраните сгенерированный код на C (копирование в буфер обмена происходит автоматически)

[2] https://fizruk.github.io/stella/playground/

2.3.2. Компиляция кода на C

Полученную программу на языке C необходимо скомпилировать вместе со средой времени исполнения и сборщиком мусора:

```
gcc -std=c11 <ИМЯ>.c stella/runtime.c stella/gc.c -o <ИМЯ>
```

При сборке можно указывать флаги, влияющие на отладочную печать и печать статистики среды исполнения:

- `STELLA_DEBUG` — включить отладочную печать
- `STELLA_GC_STATS` — печатать статистику работы сборщика мусора при завершении программы
- `STELLA_RUNTIME_STATS` — печатать статистику работы среды времени исполнения Stella при завершении программы

Например, следующая команда включает все флаги:

```
gcc -std=c11 \
  -DSTELLA_DEBUG -DSTELLA_GC_STATS -DSTELLA_RUNTIME_STATS \
  <ИМЯ>.c stella/runtime.c stella/gc.c -o <ИМЯ>
```

2.3.3. Запуск программы

Скомпилированная программа на языке Stella ожидает на вход натуральное число в текстовом представлении из стандартного потока ввода и, по окончанию работы логики программы, печатает результат в стандартный поток вывода.

Например:

```
// программа на Stella
language core;

// a constant function, specialized to Nat
fn Nat2Nat::const(f : fn(Nat) -> Nat) -> (fn(Nat) -> (fn(Nat) -> Nat)) {
  return fn(x : Nat) { return f }
}

// addition of natural numbers
fn Nat::add(n : Nat) -> (fn(Nat) -> Nat) {
  return fn(m : Nat) {
    return Nat::rec(n, m, fn(i : Nat) {
      return fn(r : Nat) { return succ(r) } })
  }
}

// multiplication of natural numbers
fn Nat::mul(n : Nat) -> (fn(Nat) -> Nat) {
  return fn(m : Nat) {
    return Nat::rec(n, 0, Nat2Nat::const(Nat::add(m)))
  }
}

// factorial via primitive recursion
fn factorial(n : Nat) -> Nat {
  return Nat::rec(n, succ(0), fn(i : Nat) {
    return fn(r : Nat) {
    return Nat::mul(r)(succ(i))  // r := r * (i + 1)
  } })
}
```

```
31
32 fn main(n : Nat) -> Nat {
33   return factorial(n)
34 }
```

```
# запуск скомпилированной программы с входом 10
echo 10 | ./factorial
```

```
3628800

------------------------------------------------------------
Garbage collector (GC) statistics:
Total memory allocation: 415,417,576 bytes (25,963,563 objects)
Maximum residency:       415,417,576 bytes (25,963,563 objects)
Total memory use:        64,397,736 reads and 0 writes

------------------------------------------------------------
Stella runtime statistics:
Total allocated fields in Stella objects: 25,963,634 fields
```

Список литературы

[1] Andrew W Appel. *Modern compiler implementation in ML*. Cambridge university press, 1998.