Санкт-Петербургский государственный университет
Математико-механический факультет

Кафедра системного программирования

Шашкова Елизавета Михайловна

Сборщик мусора для языка Objective Caml в инфраструктуре LLVM

Курсовая работа

Научный руководитель:
к. ф.-м. н. Булычев Д. Ю.

Санкт-Петербург
2014

Оглавление

Введение **3**

1. Представление данных в OCaml **4**

2. Обход объектов **6**

3. Результаты **9**

Заключение **14**

Введение

Сборка мусора — одна из форм автоматического управления памятью. Специальный компонент поддержки среды времени исполнения, называемый сборщиком мусора, автоматически освобождает память, удаляя объекты, которые больше не являются доступными для использования. Сборка мусора освобождает программиста от необходимости вручную выделять и освобождать память и позволяет сконцентрироваться на выполнении других задач. В большинстве современных языков все чаще происходит отказ от предоставления явных средств управления памятью в пользу сборки мусора.

Сборка мусора тесно связана с реализацией компилятора: например, он должен предоставлять сборщику мусора способ определить адреса корневых объектов во время исполнения. Один из основных подходов к созданию компиляторов — использование инфраструктур. Одной из таких инфраструктур построения компиляторов является LLVM (Low Level Virtual Machine) [1].

LLVM предназначена для анализа, трансформации и оптимизации программ. В основе LLVM лежит промежуточное представление кода (Intermediate Representation, IR), над которым можно производить трансформации во время компиляции и компоновки. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ.

Objective Caml (OCaml) [2] — современный объектно-ориентированный язык функционального программирования общего назначения. Для языка OCaml существует единственный компилятор от разработчиков самого языка, однако он содержит довольно много платформозависимого кода. LLVM улучшает переносимость кода между различными платформами и тем самым упрощает процесс построения собственного компилятора. Однако язык OCaml требует сборки мусора и реализованный для него бэкенд рассчитывает на наличие сборщика мусора. Но в инфраструктуре LLVM нет стандартного способа осуществления сборки мусора, и разработчики предоставили возможность для создания расширения LLVM, позволяющего её реализовывать. Данное расширение было реализовано в виде подключаемого модуля [3].

Целью данной курсовой работы является проведение интеграции бэкенда для языка OCaml и расширения для сборки мусора в LLVM. Необходимо изучить внутреннее устройство объектов в языке OCaml и реализвать стадию пометки (mark) для алгоритма сборки мусора mark-and-sweep.

1. Представление данных в OCaml

Стадия пометки в алгоритме сборки мусора mark-and-sweep заключается в том, чтобы обойти все "живые" (всё ещё доступные из корневого множества) объекты и пометить их. Затем на стадии очистки все объекты, которые не были помечены на данной стадии, будут считаться недоступными в программе, а значит, будут подлежать удалению.

Для реализации стадии пометки необходимо узнать внутреннее устройство объектов. Все значения в языке OCaml имеют единообразное представление: все они являются типом value языка C (это значение и макросы для работы с ним определены в файле caml/mlvalues.h). Объект типа value может являться:

- целым числом;
- указателем на блок внутри кучи;
- указателем на блоки вне кучи.

Целые числа находятся не в куче, поэтому сборщиком мусора рассматриваться не будут. Блоки, находящиеся в куче, являются основной целью работы сборщика мусора. Каждый блок содержит заголовок с информацией о длине блока и теге.

Тег определяет структуру блока. Теги со значением меньше, чем `No_scan_tag`, означают, что данный блок является корректно структурированным и может быть рекурсивно обойден сборщиком мусора. Теги со значением больше, либо равные `No_scan_tag`, означают, что содержимое данного блока должно быть проигнорировано сборщиком мусора. Внутреннее содержимое блоков соотносится с их тегами согласно следующей таблице:

Тег	Содержимое блока
от 0 до `No_scan_tag` – 1	Структурированный блок (массив объктов OCaml). Каждое поле является типом value.
`Closure_tag`	Замыкание представляет собой функциональное значение. Первое слово является указателем на участок кода, остальные являются объектами типа value.
`String_tag`	Строка символов.
`Double_tag`	Число с плавающей точкой двойной точности.
`Double_array_tag`	Массив чисел с плавающей точкой двойной точности.
`Abstract_tag`	Блок, представляющий абстрактный тип данных.
`Custom_tag`	Блок, представляющий абстрактный тип данных, с привязанными пользовательскими функциями.

Благодаря такому устройству объектов становится возможным обход объектов, доступных из живых объектов. Можно быть уверенными, что в процессе обхода мы не уйдем в область памяти, неинтересную сборщику мусора, а также не пропустим интересующие нас объекты.

2. Обход объектов

Наиболее естественным алгоритмом обхода объектов можно назвать рекурсивный алгоритм. Его описание представлено ниже.

1. Алгоритм запускается для каждого корневого объекта.

2. Проверить, что текущий объект является блоком (а, например, не целым числом).

3. Если объект является блоком, узнать значение его тега.

4. Если значение тега меньше значения `No_scan_tag`, объект является живым. Нужно проверить, что данный объект находится в куче и пометить его как живой.

5. Если значение тега больше или равно значению `No_scan_tag`, значит, объект должен игнорироваться сборщиком мусора.

6. После пометки самого объекта нужно последовательно обойти блоки, содержащиеся внутри данного блока (их количество известно из заголовка) и повторить с ними ту же операцию.

7. Отдельное внимание нужно уделить объекту с тегом `Closure_tag`, так как первое слово является указателем на код и при обходе сборщиком мусора должно игнорироваться.

Однако, рекурсивный вызов функций потребует дополнительной памяти для сохранения стека вызовов, причем размер этого стека предсказать невозможно. Поэтому данный алгоритм необходимо было превратить в нерекурсивный, с использованием собственного стека. Структуру объектов программы можно представить в виде графа, где вершины — объекты, а ребра — ссылки между объектами. Граф можно обойти двумя способами: в ширину (Breadth-first search) и в глубину(Depth-first search).

При обходе в глубину у каждого объекта поочередно просматриваются объекты, на которые он ссылается. Для каждого такого объекта помечаются все его потомки и только потом алгоритм переходит к следующему потомку текущего объекта. Код Mark описывает данный алгоритм. Здесь $push()$ и $pop()$ — операции работы со стеком.

При обходе в ширину объекты рассматриваются в порядке возрастания расстояния от них до коневого объекта в графе объектов. Данный алгоритм описывается так же, как и алгоритм Mark, с тем лишь отличием, что вместо стека

Algorithm 1 Mark

```
1: for root ∈ root_objects do
2:     if is_object(root) then
3:         if tag(root) < No_scan_tag then
4:             if root in heap then
5:                 mark(root)
6:             end if
7:             push(root)
8:             while object ← pop() do
9:                 size ← size(object)
10:                if tag(object) = Closure_tag then
11:                    i = 1
12:                else
13:                    i = 0
14:                end if
15:                for i < size do
16:                    child ← Field(object, i)
17:                    if is_object(child) and tag(child) < No_scan_tag then
18:                        if child in heap and not marked(child) then
19:                            mark(child)
20:                            push(child)
21:                        end if
22:                    end if
23:                end for
24:            end while
25:        end if
26:    end if
27: end for
```

используется очередь, реализованная на массиве. То есть в данном псевдокоде операции $push()$ и $pop()$ — операции работы с очередью.

Обе модификации данного алгоритма (с обходом в глубину и в ширину) были реализованы итеративно с созданием стека в статической области памяти. Отсюда понятно, что размер стека в таком случае является постоянным, и может произойти переполнение стека. Для данной ситуации существует специальный алгоритм восстановления.

В случае переполнения стека нельзя допустить некорректную работу сборщика мусора. Если произойдет ошибка в работе стадии пометки, часть живых объектов не будут помечены, то есть будут считаться мертвыми и память из-под них будет преждевременно освобождена, что приведет к некорректной работе программы. Алгоритм восстановления при переполнении стека был описан в [4]. Для реализации было решено использовать модификацию этого алгоритма [5]. Этот алгоритм, работающий с ограниченной памятью, можно описать следующим образом:

1. При заполнении стека алгоритм пометки продолжает свою работу. Однако новые объекты, которые не поместились в стек, в него не добавляются, а просто игнорируются.

2. При переполнении устанавливается специальный флаг, сообщающий о том, что переполнение произошло.

3. При завершении стадии пометки проверяется состояние флага о переполнении.

4. Если переполнение произошло, значит, часть живых объектов не была помечена. Начинается последовательный обход кучи.

5. Флаг переполнения сбрасывается. Начиная с каждого помеченного (то есть точно живого) объекта кучи запускается алгоритм пометки.

6. Теперь для пометки будет рассматриваться меньше объектов и велика вероятность, что они поместятся в стек. В случае же, если переполнение произойдет снова, алгоритм повторится (снова установится флаг переполнения и т.д.).

Нужно пояснить, что данный алгоритм не будет работать бесконечно долго, так как при каждом последующем запуске стадии пометки мы помечаем как минимум $n+1$ объект из ранее не помеченных (где n — размер стека). Количество объектов в куче конечно, а значит, процесс когда-нибудь закончится. Все живые объекты будут помечены и стадия маркировки корректно выполнит свою задачу даже при условии переполнения стека.

3. Результаты

В результате был создан модуль, реализующий стадию маркировки для алгоритма сборки мусора и произведена модификация кучи для обработки переполнения стека. Реализация была выполнена на языке C.

Наиболее интересным было исследовать работу программы при переполнении стека. В качестве тестовой программы был взят парсер с C-подобного языка, написанный на языке OCaml. Он принимает на вход файл для разбора и анализирует его. Во всех экспериментах, представленных ниже, на вход подавался один и тот же файл.

Сборка мусора вызывалась автоматически, использовался алгоритм инициализации, основанный на количестве доступной свободной памяти. Согласно этому алгоритму, сборка мусора начинается, когда объем занятой памяти в куче превышает некоторый порог, заданный переменной $Threshold$. В данных экспериментах значение переменной $Threshold$ равнялось 0.75, что соответствует моментам, когда 75% объема памяти кучи заняты.

Благодаря тому, что вызов сборки мусора основывался на количестве доступной памяти, во всех экспериментах количество вызовов сборки мусора было одинаково и равнялось 1370. К тому же сборка мусора производилась на одинаковых стадиях работы программы.

На Рис. 1 представлена зависимость времени работы программы от величины стека. При размере стека, большем 8, переполнения не происходит. Из графика видно, что переполнение стека значительно, почти в 2 раза, увеличивает время работы программы. Из графика видно, что время работы равномерно убывает при изменении размера стека от 1 до 5. При дальнейшем увеличении размера стека, на графике заметна некоторая «ступень»: при увеличении размера стека от 5 до 8 время работы программы почти не изменяется. И лишь после этого оно еще раз уменьшается и перестает зависеть от дальнейшего увеличения стека.

Для исследования такого поведения, исследуем количество переполнений, происходящих при работе программы.

На Рис. 2 показана зависимость количества переполнений стека от размера стека. Из этого графика хорошо видно, почему график времени работы имеет такую неравномерную структуру, а также «ступень». Становится понятно, что время работы программы с переполнениями стека напрямую зависит от количества этих переполнений, которые происходили в процессе выполнения. И равномерное убывание времени на промежутке от 1 до 5, и постоянное время на промежутке от 5 до 8 полностью объясняется зависимостью количества переполнений, которая ведет себя аналогично.

Также были исследованы аналогичные зависимости для алгоритма марки-

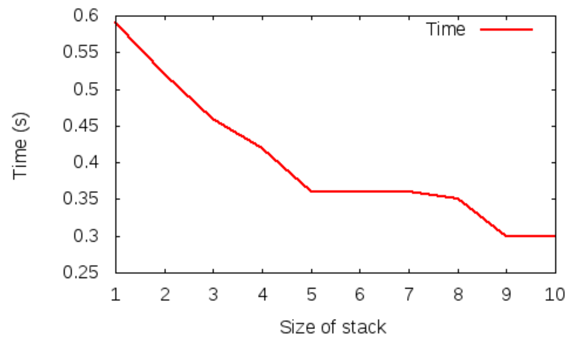

Рис. 1: Время работы программы в зависимости от величины стека с алгоритмом Mark (DFS)

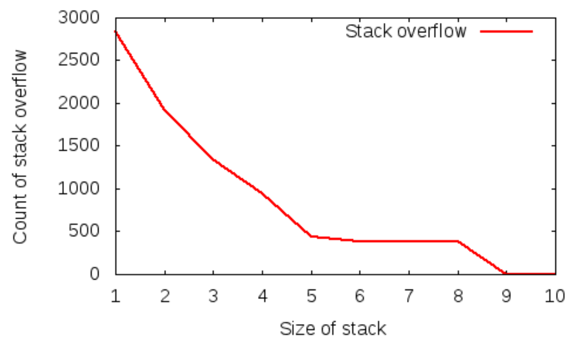

Рис. 2: Количество переполнений стека с алгоритмом Mark (DFS)

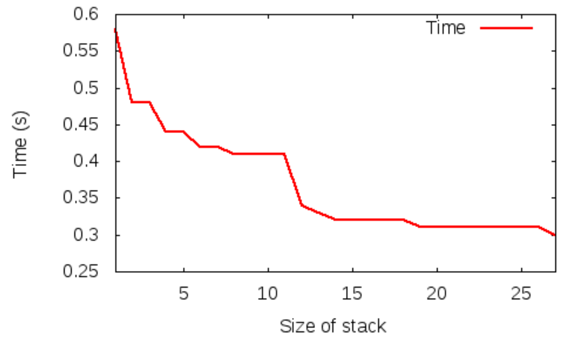

Рис. 3: Время работы программы в зависимости от величины стека с алгоритмом Mark (BFS)

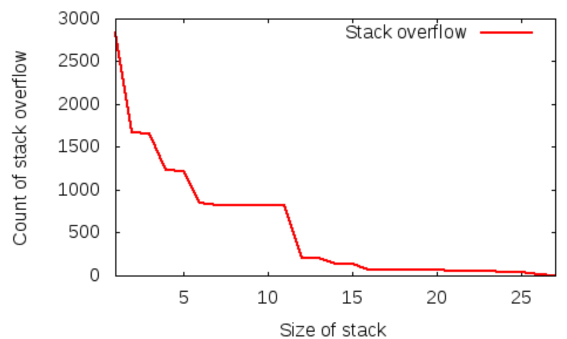

Рис. 4: Количество переполнений стека с алгоритмом Mark (BFS)

ровки с обходом объектов в ширину (Mark (BFS)).

На Рис. 3 представлена зависимость времени работы программы от величины стека. При размере стека, большем 25, переполнения не происходит. Как и в случае с алгоритмом Mark (DFS) в худшем случае (при размере стека $n = 1$) время работы программы увеличивается почти вдвое. Из особенностей графика можно отметить резкое падение времени работы при увеличении размера стека с 11 до 12. Аналогично для объяснения такой зависимости, рассмотрим поведение количества переполнений в зависимости от размера стека.

На Рис. 4 показана зависимость количества переполнений стека от размера стека с алгоритмом Mark (BFS). Как и в случае с алгоритмом Mark (DFS), очевидно, что время работы напрямую зависит от количества происходящих переполнений стека. Резкое падение при изменении размера стека с 11 на 12 также объясняется количеством переполнений.

Объясним неравномерность количества переполнений, происходящих при работе каждого из алгоритмов. Будем называть *стековой глубиной* корневого объекта максимальную заполненность стека при обходе потомков данного корневого объекта.

На графиках Рис. 5 и Рис. 6 показано количество корневых объектов в зависимости от их стековой глубины для обоих алгоритмов. Из графика Рис. 5 понятно поведение количества переполнений с алгоритмом Mark (DFS). Количество корневых объктов со стековой глубиной от 1 до 5 равномерно убывает, как и убывает количество переполнений. «Ступень» в количестве переполнений также связана с тем, что количество корневых объектов стековой глубины больше 5 очень мало по сравнению с объектами меньшей глубины. Поэтому количество переполнений почти не изменяется и убывает, только когда переполнения перестают возникать вовсе.

Аналогично для алгоритма Mark (BFS) из Рис. 6 видно, что количество корневых объектов со стековой глубиной от 8 до 11 и больше 14 мало по сравнению с другими объектами, чем и объясняются постоянные промежутки на графике количества переполнений (Рис. 4).

Стоит обратить отдельное внимание на величину стека, при которой происходят переполнения в обоих алгоритмах. Как говорилось ранее, с использованием алгоритма Mark (BFS) стек перестает переполнятся при размере, большем 9. С алгоритмом Mark (DFS) переполнения происходят вплоть до размера, равного 25. То есть при переходе к другому алгоритму размер стека увеличивается почти в 3 раза. Скорее всего, это связано со структурой объектов в тестовой программе.

Граф, описывающий структуру наших объектов, является лесом из почти деревьев. Под почти деревьями будем понимать традиционные деревья, определенные в теории графов, с тем лишь отличием, что они могут содержать петли

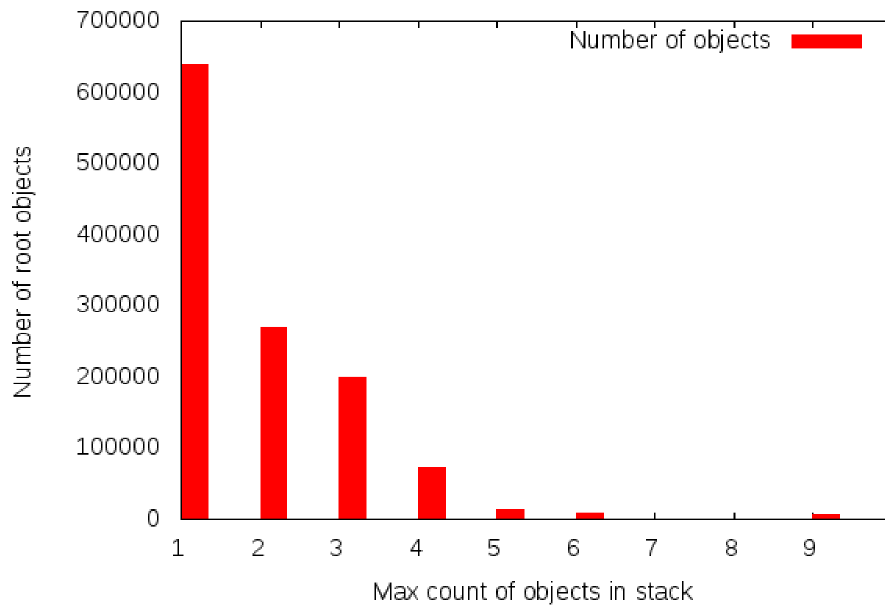

Рис. 5: Количество корневых объектов в зависимости от глубины стека с алгоритмом Mark (DFS)

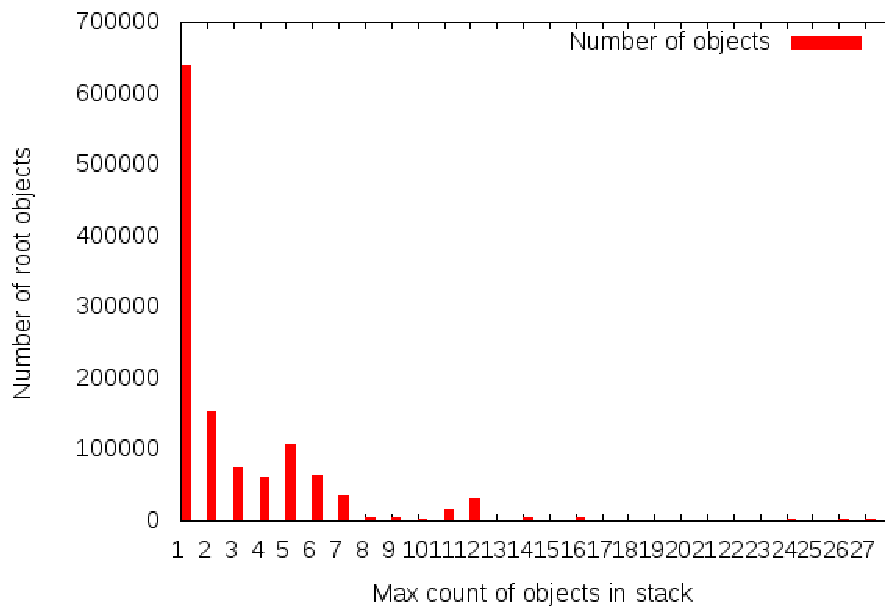

Рис. 6: Количество корневых объектов в зависимости от глубины стека с алгоритмом Mark (BFS)

(вершины, связанные ребром сами с собой). Появление петель связано с наличием замыканий (closure) в множестве наших объектов. Однако наличие петель не влияет на глубину и ширину дерева, поэтому мы можем считать наши графы деревьями.

Такая значительная разница в размерах стека означает, что у многих деревьев их ширина больше глубины. Анализ объектов в тестовой программе показал, что количество потомков у объекта не превышает трех. Даже при небольшой глубине дерева количество объектов на каждом уровне растет довольно быстро, что и приводит к увеличению размера используемого стека при обходе в ширину, в отличие от обхода в глубину.

Заключение

В результате работы была проведена интеграция бэкенда для языка OCaml и расширения для сборки мусора в LLVM. Было изучено внутреннее представление объектов в языке OCaml и реализованы два итеративных алгоритма обхода объектов в стадии пометки сборщика мусора. Результат работы был протестирован на простейших программах и корректно работает на более сложных примерах.

Список литературы

[1] Lattner Chris. LLVM: An Infrastructure for Multi-Stage Optimization. — Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[2] Ocaml language family official website. — http://ocaml.org/.

[3] А.В. Самофалов. Принципы организации сборщика мусора в инфраструктуре LLVM. — СПбГУ. Курсовая работа, 2014.

[4] Кнут Д. Искусство программирования, том 1. Основные алгоритмы. — М.: «Вильямс», 2006.

[5] О.А. Плисс. Материалы к летней школе «Управление памятью». — СПбГУ, 2013.