Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Ярков Иван Сергеевич

Escape-анализ в GoLand

Курсовая работа

Научный руководитель:
ст. преп. кафедры системного программирования СПбГУ Я. А. Кириленко

Консультант:
программист JetBrains А.С. Хвастунов

Санкт-Петербург
2019

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор **6**

2.1. Escape-анализ 6

2.2. Оптимизация подстановки тел функций 6

2.3. Существующие решения 7

3. Правила Escape-анализа **8**

3.1. Пример правила 8

3.2. Задокументированные случаи 10

4. Правила оптимизации подстановки тел функций **11**

4.1. Пример правила 11

4.2. Задокументированные случаи 12

5. Тестирующая программа **14**

5.1. Требования к тестирующей программе 14

5.2. Реализация тестирующей программы 14

Заключение **16**

5.3. Текущие результаты 16

5.4. Развитие работы 16

Список литературы **17**

Введение

К современным компиляторам предъявляются высокие требования к качеству сгенерированного кода. Для достижения этой цели, компиляторы применяют различные техники анализа кода. Одной из таких техник является Escape-анализ. Данный анализ решает, будет ли память под переменную размещена на куче или на стеке [6], таким образом влияя на производительность программы. Также, с Escape-анализом тесно связана оптимизация подстановки тел функций в места их вызова. Она облегчает проведение такого анализа и дает больше возможностей разместить память под переменные на стеке. Одним из языков в котором применяется Escape-анализ является язык программирования Go [3].

JetBrains GoLand IDE [5] — это кроссплатформенная интегрированная среда разработки для программ на языке Go. Среда отображает различную полезную информацию связанную с исходным кодом программы на языке Go. К сожалению, в настоящее время среда не может показывать в интерактивном режиме тип размещения переменной в памяти и, соответственно не может предлагать исправление ведущее к более эффективному размещению памяти под переменную. В большинстве случаев, размещения переменных на куче менее эффективны, чем на стеке, так как повышается нагрузка на сборщик мусора , что влияет на производительность программ. Разработка инструмента который будет помогать находить такие ситуации стала причиной начать данную работу.

Задачу разработки инструмента, способного показывать такую информацию и предлагать исправление ведущее к более эффективному размещению памяти под переменную, можно разбить на два этапа. Первый этап является исследованием того, как проходит Escape-анализ в компиляторе языка Go. Необходимо провести документирование поведения компилятора при таком анализе, а также составить тестовые наборы состоящие из различных программ на языке Go. Предварительно, необходимо также задокументировать и составить подобные тестовые

наборы для оптимизации подстановки тел функций в места их вызова, так как данная оптимизация проходит перед Escape-анализом и непосредственно влияет насколько успешно этот анализ сможет пройти. Дополнительно, необходимо написать тестирующую программу, которая позволит генерировать информацию по оптимизации подстановки тел функций для указанного компилятора и набора тестов, а также проверять как отличается данная информация для разных версий компилятора. Второй этап является написанием собственного статического анализатора исходного кода с помощью полученной информации из первого этапа. Написанная тестирующая программа поможет понять как изменяется оптимизация подстановки тел функций в следующих версиях компилятора, что поможет обновлять инструмент в GoLand в соответствии с ними. Настоящая работа является реализацией первого этапа.

1. Постановка задачи

Целью работы является подготовка к написанию статического анализатора для GoLand IDE. Для ее достижения необходимо предпринять следующие шаги:

1. Документировать правила Escape-анализа и составить тестовые наборы для них
2. Документировать правила оптимизации подстановки тел функций и составить тестовые наборы для них
3. Реализовать тестирующую программу для полученных тестовых наборов

2. Обзор

2.1. Escape-анализ

Escape-анализ [2] - это техника анализа кода, позволяющая определить определить область достижимости для указателя какого-то объекта во время компиляции. Анализ выясняет, может ли быть получен доступ к памяти под переменную с помощью указателей на нее из функций вызывающих ту, где эта переменная объявляется. Если анализ может доказать, что доступа к переменной из внешних вызывающих функций нет, то память под такую переменную может быть безопасно размещена на стеке, что является более производительным решением по сравнению с ее размещение на куче. Таким образом, Escape-анализ применим только к языкам программирования с автоматическим управлением памятью. Escape-анализ проходит по разному, в зависимости от языка программирования на исходном коде которого проводится анализ. Различные конструкции и синтаксические возможности языка могут осложнять анализ. Текущая работа ориентируется на Escape-анализ языка Go.

2.2. Оптимизация подстановки тел функций

Оптимизация подстановки тел функций [4] в места их вызова позволяет уменьшить накладные расходы на вызов функций и предоставить более широкие возможности для других оптимизаций и анализов. Escape-анализ является примером, когда проведение данной оптимизации перед анализом предоставляет новые возможности для него, так как подстановка тел функций упрощает структуру исходного кода программы. Компилятор языка Go имеет внутреннюю стоимостную модель по которой он оценивает насколько эффективно совершать подстановку тел функций. Данная модель учитывает различные факторы, включающие в себя вызовы специальных функций, использование синтаксических конструкций языка Go.

2.3. Существующие решения

На данный момент, единственным способом получить информацию о результатах Escape-анализа и оптимизации подстановки тел функций является передача компилятору специальных опций. Компилятор выводит данную информацию в стандартный поток ошибок.

Информация по Escape-анализу поясняет то, какой тип размещения памяти под переменную был совершен. Если происходит размещение на куче, вывод показывает причину такого размещения. В случае размещения памяти под переменную на стеке, дополнительная информация не выводится. Информация по оптимизации подстановки тел функций указывает на то, какие функции смогли быть подставлены в места их вызова, а какие нет. В случае когда такая подстановка невозможна, компилятор выводит поясняющее сообщение.

Использование данного подхода сопряжено с рядом проблем. Учитывая, что инструмент должен предлагать исправление ведущее к более эффективному размещению памяти под переменную, необходимо проходить по потоку управления программы, чтобы убедится, что такое исправление возможно и целесообразно. В этом отладочный вывод компилятора не может помочь, так как информации из него недостаточно, что ведет к необходимости написания собственного статического анализатора. Также, вызов компилятора при каждом изменении исходного кода программы будет являться причиной больших накладных расходов ресурсов компьютера. Вызов компилятора - это запуск отдельного процесса. Он не может быть таким же быстром, как работа с уже существующим деревом программы. На каждый вызов компилятор будет повторять то, что IDE уже сделала - лексинг, парсинг и построение потока управления. IDE нужно делать это вне зависимости от наличия анализатора, потому что это все переиспользуется в других местах. Результат работы компилятора скорее всего не сможет быть переиспользован.

3. Правила Escape-анализа

3.1. Пример правила

Задокументированное правило должно иметь текстовое описание и набор тестовых случаев, позволяющих подтвердить данное правило. Тест должен содержать исходный код программы на языке Go, вывод отладочной информации компилятора по Escape-анализу и пояснение такого вывода.

Как пример, рассмотрим правило описывающее поведение анализа, когда происходит присваивание элементу среза, структуре позволяющей работать с диапазоном массива. Данный пример раскроет структуру правил, по которой производилось документирование поведения Escape-анализа. Сформулируем само правило для данного случая.

Пусть ”st” - это примитивный элемент или структура типа ”T”, ”l” - это список всех указателей, которые могут быть достигнуты через поля структуры ”T” (пуст, если ”st” - это примитивный тип), ”v” - это список элементов, которые могут быть достигнуты через указатели из списка ”l”, ”sl” - это срез содержащий элементы типа ”*T”. Если происходит присваивание указателя на элемент ”st” одному из элементов среза ”sl”, то память под все элементы из списка ”v” будет размещена на куче.

Приведем один из тестовых примеров к данному правилу, который подтверждает вышесказанное. Данный пример демонстрирует, что присвоение указателя на элемент ”st” элементу среза ”sl” ведет к размещению памяти под переменную ”st” на куче. Также, происходит размещение на куче всех элементов, которые можно достигнуть через поля структуры ”st”, то есть память под переменную ”a” будет также размещена на куче. Взглянув на 3 и 9 строки отладочного вывода, можно увидеть, что память под ”a” и ”st” была размещена на куче.

```
1  package main
2
3  type S struct{
4          ptr *int
5  }
6
7  func main() {
8          a := 0
9          st := S{&a}
10         sl := make([]*S, 1)
11         sl[0] = &st
12 }
```

Рис.1: Исходный код теста

```
1  ./main.go:11:10: &st escapes to heap
2  ./main.go:11:10:  from sl[0] (slice-element-equals) at ./main.go:11:8
3  ./main.go:9:2: moved to heap: st
4  ./main.go:9:10: &a escapes to heap
5  ./main.go:9:10:   from S literal (struct literal element) at ./main.go:9:9
6  ./main.go:9:10:   from st (assigned) at ./main.go:9:5
7  ./main.go:9:10:   from &st (address-of) at ./main.go:11:10
8  ./main.go:9:10:   from sl[0] (slice-element-equals) at ./main.go:11:8
9  ./main.go:8:2: moved to heap: a
10 ./main.go:10:12: main make([]*S, 1) does not escape
```

Рис.2: Отладочный вывод компилятора по Escape-анализу

3.2. Задокументированные случаи

В ходе проведения данной работы были задокументированы следующие случаи, которые покрывают различные конструкции языка Go и составлены тестовые наборы к ним:

1. Возврат и передача указателей из функций и в них
2. Глобальные переменные
3. Непрямое присваивание полям структур
4. Интерфейсы
5. Каналы
6. Горутины
7. Массивы
8. Срезы
9. Вызовы функций по указателям
10. Анонимные функции

4. Правила оптимизации подстановки тел функций

4.1. Пример правила

Аналогично случаю с правилами для Escape-анализа, задокументированное правило должно иметь текстовое описание и набор тестовых случаев, позволяющих подтвердить данное правило. Тест должен содержать исходный код программы на языке Go, вывод отладочной информации компилятора по оптимизации подстановки тел функций и пояснение такого вывода.

Тесты для данной оптимизации разделены на две группы. Первая покрывает случаи, когда подстановка тела функции не может быть проведена, вторая оценивает стоимость функций с различными языковыми конструкциями для данной оптимизации.

Рассмотрим, как пример, случай, когда подстановка тела функции не может быть проведена. Если функция содержит цикл, то подстановка ее тела невозможна. Взглянув на строку в отладочном выводе, можно убедится, что данная подстановка невозможна. Эта строка говорит, что если в абстрактном синтаксическом дереве [1] функции есть узел являющийся циклом, то эту функцию нельзя встраивать в места ее вызова.

```
1  package main
2
3  func main() {
4          For()
5  }
6
7  func For() {
8          for false {
9          }
10 }
```

Рис.3: Исходный код теста

```
1  ./main.go:7:6: cannot inline For: unhandled op FOR
```

Рис.4: Отладочный вывод компилятора по оптимизации подстановки тел функций

4.2. Задокументированные случаи

В ходе проведения данной работы были задокументированы следующие случаи, которые покрывают различные конструкции языка Go, когда подстановка тела функции не может быть проведена:

1. Switch оператор
2. Интерфейсы
3. Анонимные функции
4. Определения новых типов
5. Аннотации функций
6. Циклы
7. Горутины
8. Рекурсивные вызовы функций
9. Вызов специальных функций

Также, было описано как языковые конструкции Go влияют на стоимость функции для данной оптимизации. Данная стоимость высчитывается через число узлов в абстрактном синтаксическом дереве функции. Были задокументированы следующие случаи:

1. Объявления
2. Присваивания
3. Вызовы других функций
4. Условные операторы
5. Обращения к массивам и срезам

5. Тестирующая программа

5.1. Требования к тестирующей программе

К тестирующей программе были поставлены следующие требования:

- Программа должна уметь генерировать информацию по оптимизации подстановки тел функций для каждой из функций с помощью указанного компилятора для указанного набора тестов
- Программа должна уметь сравнивать информацию сгенерированную двумя разными версиями компилятора Go

Тестирующая программа должна быть реализована в виде инструмента командной строки со следующим форматом:

```
verifier [-g] path/to/go/compiler path/to/tests
```

Переданная опция **-g** будет определять запущен ли режим генерации или сравнения.

5.2. Реализация тестирующей программы

Языком реализации тестирующего инструмента был выбран язык Go, так как его средств достаточно для создания инструмента командной строки и автор работы имеет опыт работы с данным языком, что позволяет не тратить дополнительное время на изучение другого.

В режиме генерации, инструмент проверяет корректность переданных аргументов, рекурсивно обходит директории с тестами по переданному пути, запускает на каждом тестовом файле с исходным кодом компилятор, получая имена функций и соответствующие результаты проведения оптимизации подстановки тел функций. Затем, к соответствующим функциям в исходном коде добавляется полученная информация в виде комментариев в фиксированном формате. Формат содержит информацию, возможна ли подстановка функции, ее стоимость или причина по которой подстановка тела функции невозможна.

Тестовые наборы, полученные во время документирования поведения оптимизации подстановка тел функций, были использованы для проверки корректности данной программы.

Заключение

5.3. Текущие результаты

В рамках курсовой работы были решены следующие задачи:

1. Документированы правила Escape-анализа и составлены тестовые наборы для них
2. Документированы правила оптимизации подстановки тел функций и составлены тестовые наборы для них
3. Реализована тестирующая программа для полученных тестовых наборов

5.4. Развитие работы

Следующий этапом является непосредственная разработка инструмента, используя полученные результаты, и внедрение его в коммерческий проект GoLand IDE.

Список литературы

[1] AST. (дата обращения: 16.05.2019). — URL: https://ru.wikipedia.org/wiki/Абстрактное_синтаксическое_дерево.

[2] Escape-analysis. (дата обращения: 10.05.2019). — URL: https://en.wikipedia.org/wiki/Escape_analysis.

[3] Go. (дата обращения: 10.05.2019). — URL: https://golang.org.

[4] Inlining. (дата обращения: 11.05.2019). — URL: https://en.wikipedia.org/wiki/Inline_expansion.

[5] JetBrains. GoLand. (дата обращения: 02.05.2019). — URL: https://www.jetbrains.com/go.

[6] Stack or heap. (дата обращения: 06.05.2019). — URL: https://golang.org/doc/faq#stack_or_heap.