Оглавление

Введение

В современном мире оптимизация программ является неотъемлемой частью процесса разработки программного обеспечения. Существует огромный спектр методов оптимизации. Одни делают оптимизации на уровне языка, другие оптимизируют скомпилированную программу. Часть предназначена для оптимизации вызовов функций, другие же специализируются на работе с многопоточными приложениями. На фоне данного многообразия, выделяется подход, называемый специализацией[link]. Программный компонент, производящий специализацию, называется специализатором (specializer) или частичным вычислителем (partial evaluator). Специализатор spec принимает на вход программу p и часть её входных данных in_1. Результатом работы специализатора является программа p_{spec}. Она представляет из себя оптимизированную версию программы p. Программа p_{spec} принимает на вход оставшуюся часть входных данных программы p и выдаёт такой же результат, что и программа p на всех входных данных.

$$\begin{array}{llcl} [\![spec]\!]_{L_2} & [p, in_1] & = & p_{spec} \\ [\![p_{spec}]\!]_{L_1} & [in_2, \ldots] & = & out \end{array}$$

Выполнение данного свойства и является определением специализатора. Важной отличительной чертой специализации, является то, что для применения метода необходимо знать часть входных данных программы. Это позволяет гораздо сильнее оптимизировать программы, но сужает сферу применения этого метода.

В 1973 году Футамура (E. Futamura, [link]) предложил использовать специализацию для несколько других целей и сформулировал проекции Футамуры-Ершова-Турчина. Идея проекций заключается в следующем. Положим, нам даны язык L, самоприменимый специализатор для языка L и интерпретатор некоторого языка R на языке L. Самоприменимость специализатора значит, что он может принимать себя в качестве входных данных. В частности, из этого следует, что язык реализации интерпретатора должен совпадать с языком, который он может принимать на вход. Первая проекция говорит о том, что при специализации интерпретатора на программу на языке R получится семантически эквивалентная программа, но на языке L. Вторая проекция говорит о том, что при самоприменении специализатора на интерпретатор получится компилятор из языка R в язык L.

$$\begin{array}{lllcl} I & [\![spec]\!]_L & [int, source] & = & target \\ II & [\![spec]\!]_L & [spec, int] & = & comp \\ III & [\![spec]\!]_L & [spec, spec] & = & cogen \end{array}$$

В настоящее время количество языков программирования неустанно растёт. Многие люди стремятся создать свой язык программирования, удобный для них или для решения их задач. Данная техника могла бы значительно упростить разработку новых

языков, однако не сыскала достаточной популярности. Данное исследование призвано продемонстрировать, что проекции Футамуры могут быть успешно применены на практике.

1. Цели и задачи

Цель и задачи нужно подкорректировать

1.1. Цель

Исследование возможностей специализатора машинного кода

1.2. Задачи

- Изучить существующие подходы и алгоритмы специализации для низкоуровневых языков программирования
- Разработать архитектуру специализатора с учётом рассмотреных подходов и особенностей языка специалиации
- Упростить структуру специализатора
- Исследовать возможности полученного специазатора

2. Существующие подходы для специализации машинного кода

Тут нужно рассказать про Partial evaluaion of machine code и может что-нибудь ещё

3. Особености специализации машинного кода

Специализация машинного кода отличается от специализации императивных или функциональных языков програмирования. В данной главе описываются проблемы и особенности возникающие при специализации машинного кода и методы их решения.

3.1. Частое использование регистров

При offline специализации программ на императивном языке программирования, BTA в самом начале классифцирует все переменные на статические и динамические. Такая классификация загрубляет результат в случае, если в одной переменной могут оказаться как динамические, так и статические значения. Нетрудно привести примеры, где из-за подобного загрубления большая часть программы становится динамической и специализация не приводит к желаемым результатам. Однако, в обычных программах зачастую используется множество переменных и значения, которые может принимать каждая из них, логически связаны. Поэтому данное загрубление либо никак не влияет на результат специализации, либо влияет незначительно.

При специализации машинного кода возникает обратная ситуация. Рассмотрим следующий фрагмент кода.

```
1 mov %esi %eax
2 mov 4 %eax
```

Пусть значение регистра *esi* было динамическим. Тогда BTA классифицирует первую инструкцию как динамическую. Её результат записывается в регистр *eax*, из-за чего он тоже становится динамическим. В случае offline специализации, данный регистр останется динамическим навсегда. Так, после выполнения второй инструкции значение регистра *eax* останется динамическим, а не статическим со значением 4. Регистр *eax* используется довольно часто. В частности, он используется для передачи возвращаемого значения из функций. Классификация его как динамический приведёт к тому, что все возвращаемые значения станут динамическими. Также, ввиду своей популярности, данный регистр имеет большой шанс, что в него положат динамическое значение. В результате получается, что в подавляющем большинстве программ данный регистр будет классифицирован как динамический и, как следствие, существенная часть программы станет динамической, что приведёт к недостаточной специализации.

Чтобы решить данную проблему, необходимо использовать другой, более сложный, метод специализации - online специализация. Его отличие от offline специализации заключается в том, что классификация регистров и памяти может меняться не протяжении всей программы, в зависимости от того, какое значение там лежит. В БАЗОВОЙ СТАТЬЕ был использован online алгоритм, который в

самом начале разбивал инструкции на статические, динамические и lifted, но классификация регистров и памяти не была жёстко зафиксирована. В данной работе был выбран ещё более гибкий способ специализации. Каждый регистр и память имеют маркер, показывающий, являются ли они статическими или динамическими. Изначально статические и данимические значения имеют соответсвующие маркеры. При специализации очередной инструкции, её классифкация определяется на основе маркеров её параметров. Таким образом, не только регистры и память могут менять свою классификацию, но и конкретные инструкции могут быть классифицированы по-разному, в зависимотси от выставленных маркеров.

Возвращаясь к примеру кода, теперь, после выполнения второй инструкции, на регистре *eax* будет выставлен флаг, что его значение статическое и также будет записано само это значение.

3.2. Комплексные инструкции

При специализации машинного кода необходимо обратить внимание на ряд специфических инструкций. Рассмотрим особенности данных инструкций на примере инструкции *push*. Дело в том, что данная инструкция состоит из двух независимых инструкций. Первая инструкция кладёт значение на вершину стека, а вторая - передвигает регистр *esp*. Рассмотрим следующий фрагмент.

```
1 push %esi
2 push 4
3 pop %eax
```

Пусть значение регистра *esi* является динамическим. В таком случае, вся инструкция будет классифицирована как динамическая. Так как в регистр *esp* записывается один из результатов испольнения данной инструкции, его значение станет динамическим. Вторая инструкция окажется динамической из-за динамического регистра *esp*. Тертья инструкция тоже будет классифицирована как динамическая и после её выполнения значение *eax* будет динамическим. Ввиду того, что программы довольно часто используют программный стек, данная проблема приведёт к тому, что большая часть программы станет динамической.

В БАЗОВОЙ СТАТЬЕ для решения этой проблемы был использован следующий алгоритм. На начальном этапе специализации подобные комплексные инструкции заменялись на две эквивалентные. Например, инструкция push заменялась на *mov* на вершину стека и *add* к вершине стека. Такое решение вызвано необходимостью последующего конвертирования в QFBV формулы. В [данной работе|разработаном спеуиализаторе]??? применялось несколько иное решение. Для каждой машинной инструкции испоьзуется конкретная фунция, отвечающая за её специализацию. Специфическая обработка реализована уже непосредственно в этой

функции. Данное решение позволило несколько упростить алгоритм специализации.

После данных изменений, регистр *esp* остаётся статическим после выполнения всех вышеперечисленных инструкций и в результате их выполнения в регистре *eax* окажется статическое значение 4.

3.3. Константные значения времени исполнения

Ещё одна проблема, специфическая для специализации машинного кода, возникает при работе с константными значениями времени исполенения программы. Такими константами, например, являются начало стека, адреса динамически выделенной памяти и адреса из статической области программы. С одной стороны, такие адреса являются известными и неизменными в процессе исполнения программы, поэтому для успешной работы специализатор должен рассматривать их как статические. С другой стороны, их конкретные значения становятся известны только в процессе исполения программы и неизвестны во время её специализации. Тривиальным решением было бы классифицировать такие значения, как динамические. Это будет отражать тот факт, что мы действительно не знаем значения данных адресов. Но это приводт к другим сложностям. Например, дно стека является начальным значением регистра *esp*. Как уже обсуждалось выше, классификация данного регистра как динамический недопустима.

Выходом из данной ситуации является использование символьных вычислений. Это решение было использовано в БАЗОВОЙ СТАТЬЕ и в данной работе оно было повторено. Все значения необходимо разделить на относительную часть и символьную часть. Если значение является адресом, его символьная часть отвечает за абсолютную константу времени исполнения, а относительная часть отвечает за конкретное смещение относительно данной константы. Если же значение адресом не является, его символьное часть равна -1, а относительная часть и является данным значением.

Такая работа с памятью уменьшает количество возможных операций с ней, однако все валидные операции остаются доступными. Так, например, можно прибавить или вычесть константу из адреса. В этом случае изменяется абсолютное значение адреса. Можно вычислить разность адресов с одинаковым символьным значением. Разница равна разнице абсолютных значений. Результат умножения адреса на константу или суммы адресов, получить невозможно, но данные операции обычно являются невалидными. При генерации кода, символьные значения необходимо обратно перевести в некую конструкцию машинного кода. Одним из вариантов является присваивание фактических адресов символьных значений в неиспользуемые регистры и использование данных регистров вместо символьных значений. При большом числе символьных значений, можно выделить отдельную память, где будут храниться фактические ад-

реса символьных значений.

4. Упрощение структуры специализатора

Специализатор из БАЗОВОЙ СТАТЬИ предназначен для оптимизации машинного кода. Основной задачейпри реализации специализатора в данном иследоовании является его самоприменимость Требуется не только изменить язык реализации, но и поменять некоторые подходы. Об этом и пойдёт речь в данной главе.

4.1. Расшарение языка специализации

В БАЗОВОЙ СТАТЬЕ использовалось достаточно небольшое подмножество машинных инструкций. После компиляции специализатора и интерпретатора возникает гораздо большее количество инструкций, которые тоже необходимо обрабатывать. В итоговый списписок вошли инструкции: БЛА-БЛА-БЛА. Данный набор не является ограничением специализатора. Остальные инструкции не были реализованы, потому что они не являются необходимыми для самиприменения и их можно добавить при необходимости.

4.2. Отказ от QFBV формул

coming soon

4.3. Отказ от lifted инструкций

coming soon

5. Исследование возможностей специализатора

Исследование возможностей полученного специализатора выполнено на наборе контрольных тестов. Результаты тестирования позволяют продеменстрировать возможности ланного специализатора и выявить ключевые проблемы.

Интерпретация результатов теста

Результатом работы специализатора является программа на ассемблероподобном языке. Отличается он от ассемблера двумя особенностями. Во-первых, в записи инструкций входят их номер в машинном коде. Это позволяет более нагляно продемонстрировать результаты работы специализатора. Во-вторых, в данном языке присутствуют несколько инструкций, отсутсвующих в ассемблере. Они заменяют код, который должен быть сгенерирован специализатор и отражают смысл данного кода. ТУТ МОЖНО ОБЪЯСНИТЬ ВСЁ ПОДРОБНЕЕ И С ПРИМЕРАМИ

5.1. Интерпретация

При специализации программы на всех входных данных, должна получаться программа, которая возвращает результат исполнения программы. В данном ключе можно рассматривать специализатор как интерпретатор.

Первый тест представляет собой специализацию функции возведения в степень на оба её входа. Ниже представлен код данной функции на языке C. Он был скомпилирован в машинный код компилятором gcc, после чего передан на вход специализатору.

```
int pow(int a, int b) {
    if (b == 0) {
        return 1;
    }
    return pow(a, b - 1) * a;
}
```

В качестве значений a и b были переданы 3 и 5 соответственно. Результатом работы специализатора стала следующая программа.

```
Start block 926794
premov 243 , %rax
ret
```

Первая строчка полученной программы говорит о начале нового блока. Вторая строчка говорит о том, что здесь необходимо сгенерировать инструкцию, которая положит в регистр rax значение 243. Последняя строчка говорит о завершении блока. В целом, полученная программа всегда возвращает 243, что является 3^5. Стоит обратить

внимание, что при специализации такой небольшой программы необходимо генерировать новые инструкции. Это происходит из-за того, что после выполнения всех инструкций внутреннее состояние специализатора изменяется, но это никак не отображается на сгенерированной программе. Поэтому необходимо выгрузить значение данного регистра, чтобы программа действительно возвращала 243, а не была пустой. Эта проблема более ярко раскрывается в следующем тесте.

Следующий тест представляет из себя специализацию алгоритма пузырьковой сортировки на массив из случайных элементов. Ниже представлен код даннго алгоритма на языке C. Он был скомпилирован в машинный код компилятором gcc, после чего передан на вход специализатору.

```
void sort(int len, int* a) {
    for (int i = 0; i < len; ++i) {
        for (int j = i + 1; j < len; ++j) {
            if (a[i] > a[j]) {
                int k = a[i];
                a[i] = a[j];
                a[j] = k;
            }
        }
    }
}
```

Результатом данного теста является следующая программа.

```
Start block 623624
ret
```

Это пустая программа. С одной стороны функция сортировки ничего не возвращает, поэтому такой результат вполне закономерен. С другой стороны, результирующая программа должна отсортировать массив, но не делает этого. Если посмотреть на внутреннее состояние специализатора после специализации, можно увидеть, что внутри массив отсортирован. Достаточно выгрузить значения массива, чтобы получить программу, сортирующую массив. Проблема заключается в том, что не всегда понятно, какую часть внутреннего состояния стоит выгружать. Можно выгрузить всё состояние, но оно достаточно большое и зачастую требуется выгрузка лишь небольшого фрагмента внутреннего состояния. Чтобы не перегружать итоговые программы, было решено выгружать только значение регистра eax.

5.2. Работа с памятью

Данный тест представляет из себя проверку на работу памятью, а также работу с симовльными вычислениями. Программой для специализации является алгоритм решета эратосфена. Ниже представлен код даннго алгоритма на языке C.

```
char* eratosphen(int n) {
    char* a = my_malloc(n);
    for (int i = 0; i < n; ++i) {
        a[i] = 1;
    }
    a[0] = 0;
    a[1] = 0;
    for (int i = 2; i < n; ++i) {
        if (a[i]) {
            for (int j = i * i; j < n; j += i) {
                a[j] = 0;
            }
        }
    }
    return a;
}
```

Вместо привычной функции malloc тут используется функция my_malloc. Ниже приведён её код.

```
void* my_malloc(int n) {
    return malloc(n);
}
```

Дело в том, что вызов данной функции является особым случаем вызова функции, который отдельно обрабатывается. Функция my_malloc позволяет специализатору определить, что данный случай произошёл.

Результатом работы данного теста является следующая программа.

```
Start block -681660
call malloc
premov 0(1) , %rax
ret
```

Инструкция во второй строке говорит о том, что тут необходимо сделать вызов функции malloc для выделения памяти. Символ 1 отвечает за память, выделенную функцией malloc, поэтому в третьей инструкции выгружется указатель, который соответствует новой выделенной памяти. Данная память не заполняется флагами простых чисел из-за отсутсвия их выгрузки из внутреннего состояния специализвтора.

5.3. Специализация

Следующий тест представляет из себя базовый вариант специализации. Программа для специализации принимает на вход статическую длинну списка, статический список ключей, динамический список значений и статический искомый ключ. Программа ищет искомый ключ в списке ключей и выдаёт соответствующее значение из списка значений. В случае, если ключа нет в списке, программа выдаёт -1. Ниже

представлен код данной программы. Ниже представлен код данной программы на языке C.

```
int dict(int len, int* keys, int* values, int key) {
    for (int i = 0; i < len; ++i) {
        if (keys[i] == key) {
            return value[i];
        }
    }
    return -1;
}
```

Результат специализации, где длинна массива 3, список ключей - $[0, 1, 2]$, а искомый ключ - 1, выглядит так.

```
Start block 701084
mov89 %rdx  -48(0)
mov8b -48(0)  %rax
add01 4  %rax
mov8b 0(rax)  %rax
ret
```

В данной программе уже происходит работа со стеком и без символьных вычислений не обойтись. Символ 0 соответствует стеку. Таким обрзом $-48(0)$ это адрес, соответствующий адресу дна стека, минус 48 байт. Можно заметить, что вторая и третья строчки ничего не делают. Специализатор их оставил, поскольку в них происходит работа с динамическими объектами. Специализатор не способен соптимизаровать подобные моменты. В четвёртой строчке к адресу массива значений прибавляется 4 байта, то есть получается адрес следующей ячейки, и уже в пятой строчке значение этой ячейки возвращается в качестве результата. Таким образом, данная программа возвращает значение из массива значений с индексом 1. Данный результат и ожидался в качестве результата специализации.

5.4. Генерация машинного кода

Во всех предыдущих тестах результатом работы специализатора был ассемблероподобный код. Однако зачастую подобного кода оказывается недостаточно - требуется иметь возможность запустить результат специализации. В качестве исходной программы была взята вышеупомянутая программа возведения в степень(см. 6.1). В качестве статического показателя степени взята цифра 5. Основание же является динамическим. В результате получился следующий машинный код.

89 7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89 7D FC B8 01 00 00 00 0F AF 45 FC 0F AF 45 FC 0F AF 45 FC 0F AF 45 FC C3

Если данный фрагмент кода исполнить и передать ему в качестве параметра число, в ответе получится переданное число в пятой степени. Это является тем результатом, который ожидался. Стоит отметить, что была реализована лишь минимальная функциональность, необходимая для прохождения данного теста. Это связано с тем, что компиляция ассемблера в машинный код является непростой задачей, но не несёт в себе исследовательского интереса. Тем не менее, данный тест показывает, что такая компиляция возможна в рамках данного специализатора.

5.5. КМП-тест

Пусть есть наивный алгоритм поика подстроки в строке. Задачей кмп-теста является проверить, сможет ли тестируемая программа преобразовать наивный алгоритм поиска подстроки в строке в оптимальный (например КМП). Данный тест широко применяется для тестирования специализаторов, суперкомпиляторов и других методов оптимизации. Однако, далеко не каждый специализатор способен пройти данный тест. Это связано с тем, что для прохождения данного теста необходимо собирать и использовать негативную информацию. Негативной информацией называется информация о неравенстве некоторых величин. В отличии от позитивной информации, которая является информацией о равенстве некоторых величин, негативная информация используется не всегда, потому что негативная информация является более сложной для хранения и использования. Однако, в классическом труде [link] был предложен наивный алгоритм, хранящий негативную информацию внутри себя. Этот алгоритм был реализован на языке flowchart для самоприменимого специализатора на этом же языке. В данном исследовании этот алгоритм был реализован на языке C(см приложение 1).

МОЖНО ДОПИСАТЬ ПРО ПРИНЦИП РАБОТЫ АЛГОРИТМА И ПРИЧИНЫ ЕГО УСПЕШНОЙ СПЕЦИАЛИЗАЦИИ

При специализации, искомая строка являкется статической - *abac*, а строка, в которой производится поиск, - динамической. Результат специализации слишком большой, чтобы приводить его целиком, поэтому ниже приведён показательный фрагмент данного результата.

```
1 Start block −771808
2 mov89 %rsi  −88(0)
3 mov8b −88(0)  %rax
4 movb6 0(rax)  %rax
5 test %al %al
6 cjump 0x85 to 792400
7 premov 0 , %rax
8 ret
```

```
9
10 Start block 792400
11 mov8b -88(0)  %rax
12 movb6 0(rax)  %rax
13 cmp39 97  %rax
14 cjump 0x85 to 152061
15 add83 -88(0)  1
16 mov8b -88(0)  %rax
17 movb6 0(rax)  %rax
18 test %al %al
19 cjump 0x85 to 782129
20 premov 0 , %rax
21 ret
22
23 Start block 782129
24 mov8b -88(0)  %rax
25 movb6 0(rax)  %rax
26 cmp39 98  %rax
27 cjump 0x85 to 93280
28 add83 -88(0)  1
29 mov8b -88(0)  %rax
30 movb6 0(rax)  %rax
31 test %al %al
32 cjump 0x85 to 603573
33 premov 0 , %rax
34 ret
35
36 Start block 603573
37 mov8b -88(0)  %rax
38 movb6 0(rax)  %rax
39 cmp39 97  %rax
40 cjump 0x85 to 963237
41 add83 -88(0)  1
42 mov8b -88(0)  %rax
43 movb6 0(rax)  %rax
44 test %al %al
45 cjump 0x85 to -200042
46 premov 0 , %rax
47 ret
48
49 Start block -200042
50 mov8b -88(0)  %rax
51 movb6 0(rax)  %rax
52 cmp39 99  %rax
53 cjump 0x85 to 62602
54 add83 -88(0)  1
55 premov 1 , %rax
```

```
56 ret
57
58 Start block 62602
59 mov8b −88(0)  %rax
60 movb6 0(rax)  %rax
61 cmp39 98  %rax
62 cjump 0x85 to −747529
63 add83 −88(0)  1
64 mov8b −88(0)  %rax
65 movb6 0(rax)  %rax
66 test %al %al
67 cjump 0x85 to 586172
68 premov 0 , %rax
69 ret
70
71 Start block −747529
72 mov8b −88(0)  %rax
73 movb6 0(rax)  %rax
74 cmp39 97  %rax
75 cjump 0x85 to −484976
76 add83 −88(0)  1
77 mov8b −88(0)  %rax
78 movb6 0(rax)  %rax
79 test %al %al
80 cjump 0x85 to 936159
81 premov 0 , %rax
82 ret
```

В первом блоке происходит проверка на длинну динамической строки. Если код первого символа равен 0, динамическая строка закончилась и программа возаращает 0. В противном случае производится переход на следующий блок. В 13 строчке происходит сравнение первого символа динамической строки с символом a. В случае несовпадения, осуществляется условный переход. Показательным является случай, где символы равны. Алгоритм передвигает указатель динамической строки на следующий символ и проверяет, что строка не закончилась. В следующем блоке просиходит аналогичная проверка, но на символ b. Предположим, что символы опять совпали. В четвёртом блоке опять происходит проверка на сивол a. Рассмотрим вариант, что она тоже увенчалась успехом. В пятом блоке происходит проверка на символ c. Если символы совпали, алгоритм завершается с результатом 1, потому что искомая строка найдена. Посмотрим, что произойдёт в случае несовпадения. Алгоритм будет сравнивать этот же символ (символ из динамической строки с индексом 3) с символом b. Наивный алгоритм в случае несовпадения попытался бы проверить совпадение искомой строки начиная с индекса 1. Однако, это бессмысленно, потому что символ с индексом 1 уже совпал с символом b и не может равняться символу a. После этого,

наивный алгоритм попытался бы проверить совпадение строки с индекса 2. В данном случае, первое сравнение тоже бессмысленно, потому что символ с индексом 2 уже оказался равен символу a и он будет равен символу a снова. Следующим сравнением наивного алгоритма стало бы сравнение символа с индексом 3 с сиволом b. Это и является следующим шагом алгоритма, получанного после специализации. В случае совпадения, алгоритм уже сравнивает следующий символ динамической строки с символом a

ПЛОХОЙ ПРИМЕР. ЛУЧШЕ АВАВАС. ОН ДЕМОНСТРИРУЕТ НЕГАТИВНУЮ ИНФОРМАЦИЮ

5.6. Первая проекция

Как упоминалось выше, первая проекция Футамуры представляет из себя специализацию интерпретатора на программу. Для проведения данного теста было выбрано небольшое подмножество языка C и написан интерпретатор данного подмножества. Также, была написана следующая программа на данном языке.

```
int add(int a, int b){
    int c;
    c = a;
    c += b;
    return c;
}
```

Данная программа складывает два числа. При специализации интерпретатора на данную программу получается следующий результат.

```
Start block -697046
mov89 %rsi  -72(0)
call malloc
mov8b -72(0)  %rax
mov8b 0(rax)  %rax
mov89 %rax  0(2)
add83 -72(0)  4
mov8b -72(0)  %rax
mov8b 0(rax)  %rax
mov89 %rax  16(2)
add83 -72(0)  4
mov8b 0(2)  %rax
mov89 %rax  -100(0)
mov8b -100(0)  %rax
mov89 %rax  -28(0)
mov8b -28(0)  %rax
mov89 %rax  32(2)
```

```
18 mov8b 16(2)  %rax
19 mov89 %rax  -100(0)
20 mov8b -100(0)  %rax
21 mov89 %rax  -28(0)
22 mov8b 32(2)  %rcx
23 mov8b -28(0)  %rdx
24 add01 %rcx  %rdx
25 mov89 %rdx  32(2)
26 mov8b 32(2)  %rax
27 mov89 %rax  -100(0)
28 mov8b -100(0)  %rax
29 ret
```

Целью данного теста является оценить объём лишних инструкций, возникших из-за интерпретатора. Так как специализатор принимал на вход список параметров программы, итоговая программа принимает на вход список из двух элементов через регистр rsi. В третьей строчке происходит выделение памяти, которое изначально происходило в интерпретаторе для сохранения состояния программы. Строки 4-11 перекладывают значения из входного массива в выделенный массив. Этот фрагмент кода присутствует в интерпретаторе. Строки 12-17 соответствуют третьей строке изначальной программы. Действительно важными являются строки 12 и 17. Строка 12 считывает значение переменной a, а строка 17 записывает это значение в переменную c. Строки между ними появляются из-за того, что при интерпретации данное значение несколько раз возвращается из функции и кладётся в локальные переменные. Все эти операции являются динамическими и специализатор их оставляет в итоговой программе. Строки 18-25 соответсвуют четвёртой строке изначальной программы. Ключевыми являются строки 18 и 21-25. Остальные строки являются лишними и появились по вышеизложенным причинам. Наконец, строки 26-29 соответствуют пятой строке исходной программы. Тут ключевыми являются строки 26 и 29. С одной стороны может показаться, что лишнего кода получилось достаточно много. С другой стороны, в самом интерпретаторе довольно много инструкций, а в результирующую программу попала лишь небольшая часть. Также все лишние инструкции являются достаточно однообразными и их можно достаточно легко обнаружить с помощью отдельного анализа.

Выводы

Выводы