

Министерство образования и науки  
Российской Федерации  
Федеральное агентство по образованию  
Новосибирский государственный университет  
Физический факультет  
Кафедра Автоматизации Физико-Технических Исследований

Февральский отчет

Парфиненко Владимир Владимирович

**АНАЛИЗ УКАЗАТЕЛЕЙ И СИНОНИМОВ  
ДЛЯ МНОГОПОТОЧНЫХ ПРОГРАММ**

Научный руководитель  
м. н. с. ИСИ СО РАН, Павлов П. Е.

Новосибирск 2010

# Содержание

<b>1</b>	<b>Введение</b>	<b>3</b>
<b>2</b>	<b>Описание предметной области</b>	<b>4</b>
<b>3</b>	<b>Постановка задачи</b>	<b>8</b>
<b>4</b>	<b>Описание алгоритма анализа</b>	<b>10</b>
4.1	Строгая система типов языка Java . . . . .	10
4.2	Внутреннее представление и SSA-форма . . . . .	10
4.3	Операции и их семантика . . . . .	11
4.4	Тип алгоритма . . . . .	14
4.4.1	Subset-based алгоритм анализа . . . . .	14
4.4.2	Нечувствительный к потоку алгоритм анализ . . . . .	14
4.4.3	Внутрипроцедурный алгоритм анализа . . . . .	16
4.5	Описание алгоритма анализа для однопоточных программ . . .	16
4.6	Модель памяти языка Java . . . . .	18
4.7	Описание алгоритма анализа для многопоточных программ . .	18
	<b>Список литературы</b>	<b>19</b>

# 1 Введение

Средства оптимизации программ нужны для получения высокой скорости исполнения программ или улучшения других характеристик программы.

Под оптимизацией программ понимается преобразование программы в семантически эквивалентную, но более эффективную относительно некоторого заданного критерия. Преобразование программы  $A$  в программу  $B$  эквивалентно (или корректно), если из того, что программа  $A$  выполнима на некотором наборе данных, следует, что и  $B$  также выполнима на этом наборе и дает тот же результат, что и  $A$ . В общем случае задача проверки эквивалентности двух программ неразрешима, и не существует алгоритма, который по данной программе находил бы эквивалентную ей и оптимальную относительно заданного критерия [1].

Тем не менее существует набор известных оптимизирующих преобразований, таких, что корректность каждого из них гарантирует корректность их последовательного применения. И чтобы конкретное преобразование данной программы было корректным, необходимо выполнение некоторых условий. Например, чтобы иметь возможность убрать генерацию некоторой части программы, нужно быть уверенным, что управление никогда не попадет в эту часть программы. Для получения подобной информации используют результаты статического анализа. Анализируя код программы, деляются выводы о тех или иных свойствах программы, необходимых для проведения преобразования.

Статический анализ применяется не только для проверки корректности преобразований, но и в инструментах статического анализа кода программ, которые могут находить потенциальные ошибки и определять другие свойства программы без ее непосредственного исполнения.

Существует множество видов статического анализа, и одним из них является анализ указателей и синонимов.

## 2 Описание предметной области

Анализ указателей — это один из видов статического анализа, который позволяет определить на какие объекты в памяти могут указывать выражения ссылочного типа в программе, такие объекты называются целями выражения ссылочного типа. Анализ синонимов похож на анализ указателей, его целью является определение, могут ли два разных выражения ссылаться на одно и то же место в памяти (такие выражения называют синонимами).

Существует множество алгоритмов анализа указателей и синонимов, одним из них является алгоритм анализа, основанный на типах (Type-Based Alias Analysis) [2], применимый для языков со строгой типизацией. В языках со строгой типизацией любая ссылочная переменная формального типа  $T$  может указывать на любые объекты типа  $T$  или его наследников (подробнее про строгую систему типов в разделе 4.1). Простейшая реализация алгоритма основанного на типах дает результат такой, что независимо от контекста и потоков данных в программе целями выражения ссылочного типа являются все объекты совместимые по присваиванию с этим выражением. Такой алгоритм работает быстро, но обладает сравнительно низкой точностью.

Для сравнения точности алгоритмов анализа указателей нам необходимо ввести некую меру точности. В качестве простой меры точности алгоритма можно использовать усредненное количество синонимов для переменных ссылочного типа, появляющихся в программе [3]. Понятно, что для «идеального» алгоритма анализа это число будет минимальным, а для самого консервативного алгоритма максимальным.

Более точные алгоритмы анализа учитывают не только типы переменных, но и потоки данных в программе. Например, если существует только одно присваивание переменной нового объекта типа  $T$ , выделенного в куче, то можно гарантировать, что эта переменная может указывать только на этот объект. С присваиванием значения одной переменной другой переменной ситуация сложнее. Рассмотрим пример 1. Для удобства, цели указателя, то есть

---

**Пример 1** Сравнение subset-based и equality-based алгоритмов

---

```
1:  $b = \mathbf{new} \ T()$   
2:  $c = \mathbf{new} \ T()$   
3:  $a = b$   
4:  $a = c$ 
```

---

множество объектов, на которые может указывать указатель  $p$  (или переменная ссылочного типа), обозначим как  $Pts(p)$  (англ. Points-to set). Учитывая строки 1 и 2, для переменных  $b$  и  $c$  мы можем точно определить множество объектов, на которые они указывают,

$$Pts(b) = \{O_1\}, Pts(c) = \{O_2\},$$

где  $O_1$  и  $O_2$  уникальные объекты в куче. То есть мы учли поток данных от оператора `new`, вернувшего новый объект, в переменную.

Интерпретировать присваивание  $a = b$  можно двумя способами, и алгоритмы анализа разбиваются на два типа по этому признаку:

- алгоритмы первого типа накладывают ограничение  $Pts(a) \supset Pts(b)$  (subset-based алгоритмы),
- алгоритмы второго типа накладывают ограничение  $Pts(a) = Pts(b)$  (equality-based алгоритмы).

Первый тип алгоритмов более точен, в то время как второй быстрее [4]. Возвращаясь к нашему примеру, subset-based алгоритм получит, что

$$Pts(a) = \{O_1, O_2\}, Pts(b) = \{O_1\}, Pts(c) = \{O_2\},$$

а equality-based

$$Pts(a) = Pts(b) = Pts(c) = \{O_1, O_2\}.$$

Также алгоритм может учитывать поток управления в программе. Рассмотрим пример 2. Чувствительный к потоку управления анализ получит

---

## Пример 2 Сравнение чувствительного и нечувствительного к потоку управления алгоритма


---

```
1:  $a = \mathbf{new} \ T()$   
2:  $b = \mathbf{new} \ T()$   
3:  $c = \mathbf{new} \ T()$   
4:  $a = b$   
5:  $b = c$   
6:  $c = a$ 
```

---

следующие данные после 3-ей строки


строка 3 :  $Pts(a) = \{O_a\}, Pts(b) = \{O_b\}, Pts(c) = \{O_c\}$ .



Далее, при анализе 4-ой строки, алгоритм поймет что  $Pts(a) = \{O_b\}$ , фактически, не только сгенерировав информацию о том, что  $a$  может указывать на  $O_b$ , но и уничтожив информацию о том, что  $a$  может указывать на  $O_a$ . Проведя аналогичные рассуждения к концу программы получатся следующие результаты:

строка 6 :  $Pts(a) = \{O_b\}, Pts(b) = \{O_c\}, Pts(c) = \{O_b\}$ .

Получается, что такой алгоритм должен хранить информацию о состоянии целей указателей для каждой инструкции отдельно.



Нечувствительный к потоку управления анализ воспринимает программу не как последовательность инструкций, а как их неупорядоченный набор. Для указанного примера такой анализ получит следующее

$$Pts(a) = Pts(b) = Pts(c) = \{O_a, O_b, O_c\}.$$

Этот результат не является точным, но зато полученная информация верна для любой точки программы и может храниться в единственном экземпляре.

Рассмотрим, как алгоритм анализа может обрабатывать вызовы функций и процедур на примере 3. Наша задача понять, чему равно  $Pts(a)$ . Алгоритмы

---

**Пример 3** Демонстрация работы межпроцедурного алгоритма

---

```
1: function foo( $x, y$ ) {  
2:   return  $x$   
3: }  
4:  
5:  $b = \mathbf{new}$  T()  
6:  $c = \mathbf{new}$  T()  
7:  $a = \text{foo}(b, c)$ 
```

---

анализа можно разделить на две категории, в зависимости от того, как они обрабатывают вызовы функций:

- межпроцедурные алгоритмы анализа могут сначала проанализировать тело вызываемой функции, и затем учесть результат при обработке вызова,
- внутрипроцедурные алгоритмы рассматривают вызов функции в наиболее консервативном предположении: может быть возвращен либо один из параметров, либо любой глобальный объект, либо новый объект.

Понятно, что первый тип алгоритмов дает более точные результаты, а второй потребляет меньше памяти и работает быстрее [5, с. 117]. В нашем примере межпроцедурный алгоритм анализа, проанализировав функцию `foo`, запоминает, что для нее выполнено следующее условие на возвращаемое значение

$$Pts(retval) = Pts(x),$$

и тогда может сделать вывод, что

$$Pts(a) = Pts(b) = \{O_1\}.$$

В такой же ситуации внутрипроцедурный алгоритм анализа обязан сделать консервативное предположение

$$Pts(a) = Pts(b) \cup Pts(c) = \{O_1, O_2\}.$$

### 3 Постановка задачи

Целью данной работы является изучение существующих алгоритмов анализа указателей, последующая разработка внутрипроцедурного алгоритма и внутреннего представления для использования в оптимизирующем статическом компиляторе Java программ Excelsior Research Virtual Machine (Excelsior RVM) с учетом приведенных ниже требований.

Алгоритм анализа должен учитывать следующие особенности языка Java:

- наличие строгой типизации,
- отсутствие адресной арифметики,
- отсутствие указателей на указатели.

Именно эти особенности отличают язык Java от языка C при рассмотрении анализа указателей, для которого были спроектированы классические алгоритмы анализа указателей Стингарда [4] и Андерсена [5].

В связи с широким распространением многопоточных программ, алгоритм анализа необходимо адаптировать для применения к многопоточным программам согласно спецификации JVM, которая имеет строгое и подробное описание модели памяти (Java Memory Model) [6].

Кроме того внутреннее представление программы должно быть адаптировано, для хранения результатов анализа указателей, и предоставлять удобный интерфейс для получения этих результатов другими алгоритмами статического анализа.

Для достижения поставленной цели необходимо:

- изучив существующие алгоритмы анализа указателей, выбрать подходящий внутрипроцедурный алгоритм и адаптировать его для анализа многопоточных программ на языке Java,
- адаптировать существующее внутреннее представление программы для хранения результатов анализа указателей,



- реализовать алгоритм и внутреннее представление для оптимизирующего статического компилятора Java программ Excelsior RVM.

## 4 Описание алгоритма анализа

### 4.1 Строгая система типов языка Java

Так как до сих пор нет устоявшегося определения строгой типизации [TODO: cite], для определенности рассмотрим строгую систему типов языка Java.

В Java ссылочными типами являются классы, интерфейсы, массивы других типов; примитивные типы можно не рассматривать в контексте анализа указателей, так как они не могут переносить информацию о целях указателей. Все классы и массивы являются наследниками типа *java.lang.Object* (далее просто *Object*), будем считать, не ограничивая общность, что все интерфейсы также являются наследниками типа *Object*.


В Java допускается присваивание переменной только значения, имеющего такой тип данных, что существует расширяющее преобразование к типу этой переменной. Например, преобразование типа *byte* к типу *int*, преобразование ссылочного типа к его предку являются расширяющими преобразованиями, они безопасны и не требуют дополнительных действий на этапе исполнения. А преобразование типа *double* к типу *float*, преобразование типа *Object* к другому ссылочному типу являются сужающими, могут приводить к потере данных и требуют проверок на этапе исполнения.

### 4.2 Внутреннее представление и SSA-форма

В процессе компиляции программа на исходном языке программирования переводится в, так называемое, внутреннее представление программы (англ. internal representation, IR), на котором работают алгоритмы анализа и на котором проводятся все оптимизации. Будем рассматривать внутреннее представление тела функции, заданное в виде графа управления (англ. control flow graph, CFG). CFG — это ориентированный граф, в котором вершинам соответствуют последовательности операторов программы, а дугам — пере-



ходы из конца одной последовательности операторов в начало другой. Такие последовательности операторов, являющиеся вершинами, назовем линейными участками. В конце каждого линейного участка присутствует оператор перехода, который передает управление по одной из дуг, выходящих из данной вершины CFG.



Будем говорить, что программа находится в SSA-форме (Static Single Assignment), если существует не более одного присваивания каждой переменной. В SSA-форме вводится дополнительная операция слияния значений переменных, так называемая  $\phi$ -функция. Пусть в CFG существует вершина  $N$ , такая что в нее входит больше одной дуги из вершин  $N_1, \dots, N_K$ , назовем такую вершину точкой слияния. Тогда в  $N$  будут располагаться следующие вызовы  $\phi$ -функций


$$v = \phi(v_1, \dots, v_K),$$

для всех переменных  $v$  требующих слияния. Семантика данной операции заключается в присваивании переменной  $v$  значения переменной  $v_i$ , соответствующего вершине  $N_i$ , из которой управление пришло в  $N$ . В случае CFG, приведенного на рисунке 1, переменной  $a_3$  будет присвоено значение 5 или 7, если управление придет в нижнюю вершину через левую или правую вершину, соответственно.

Для преобразования внутреннего представления программы в SSA-форму и обратно существуют эффективные алгоритмы [7]. В то же время многие виды анализа и многие оптимизации проще и эффективнее проводить именно на внутреннем представлении в SSA-форме, поэтому его и будем использовать для проведения анализа указателей.



## 4.3 Операции и их семантика



Сначала определим точно, что может быть указателем в языке Java и подобных ему. В C-подобных языках допустимы переменные, поля объектов, элементы массивов, указывающие на:

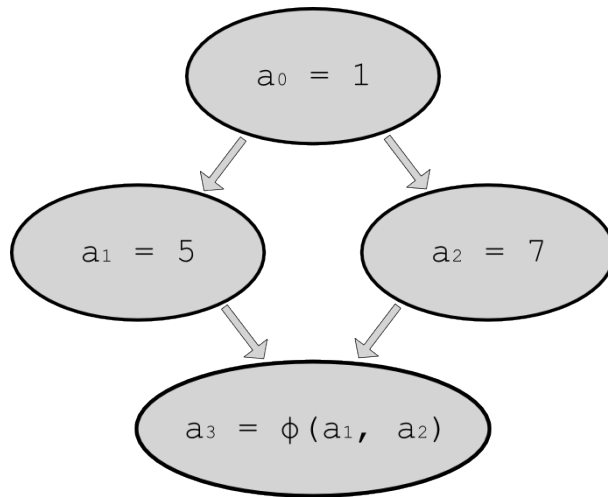


Рис. 1: Пример CFG с  $\phi$ -функцией

- объекты в куче,
- объекты на стеке,
- другие переменные,
- функции.

В Java-подобных языках ситуация существенно отличается, так как целью указателя может быть только объект, находящийся в куче. Так же отсутствует адресная арифметика (операции взятия адреса, чтения и записи по адресу), за счет чего ссылка на объект может появиться только явно, по цепочке присваиваний, начиная с создания этого объекта в куче.

Для проведения анализа указателей ~~необходимо~~ учитывать не все операции допустимые в языке, а лишь те, которые могут изменять цели указателей. ~~Необходимо~~ рассматривать лишь следующие инструкции ( $a, b, c$  — переменные или формальные параметры ссылочного типа):

Инструкция	Описание
$a = b$	присваивание значения $b$ переменной $a$
$a = \text{null}$	указание, что $a$ не указывает ни на один объект
$a = \text{new } T$	создание нового объекта типа $T$ в куче
$a = b.f$	чтение поля объекта, на который ссылается $b$
$b.f = a$	запись в поле объекта, на который ссылается $b$
$a = T.f$	чтение статического поля класса $T$
$T.f = a$	запись в статическое поле класса $T$
$a = b[\dots]$	чтение элемента массива $b$
$b[\dots] = a$	запись в элемент массива $b$
$a = (T)b$	преобразования значения переменной $b$ к типу $T$
$a = \text{foo}(b, \dots)$	вызов функции <code>foo</code> , возвращающей значение ссылочного типа
<code>foo(b, ...)</code>	вызов функции <code>foo</code> , либо не возвращающей значение, либо возвращающей значение примитивного типа

Так как для алгоритма анализа сложно определить индекс элемента при чтении и записи в массив, удобно рассматривать массив как объект с единственным полем *elements*, из которого читаются и в которое записываются все значения при обращении к элементам массива.


При преобразовании  $a = (T)b$  в случае неудачного преобразования значения переменной  $b$  к типу  $T$  будет выброшено исключение во время исполнения, поэтому в рамках анализа, можно считать что переменная  $a$  может указывать только на объекты, совместимые по присваиванию с типом  $T$ .

При вызове другой функции нужно понимать, что вызванная функция может изменять цели полей переданных параметров и полей статических переменных.

## 4.4 Тип алгоритма

Необходимо определиться с характеристиками алгоритма анализа подходящего для нашей задачи. Описание основных характеристик уже было приведено в разделе 2.


### 4.4.1 Subset-based алгоритм анализа



При выборе между subset-based и equality-based алгоритмом анализа, необходимо выбрать, что важнее для конечного алгоритма: точность или скорость работы, соответственно. Проведенные эксперименты показывают [8], что на небольших программах (до 3000 строк), время работы обоих алгоритмов анализа примерно одинаково, но точность у subset-based алгоритма значительно выше. Забегая вперед, отмечу, что алгоритм будет применяться для внутрепроцедурного анализа, и поэтому, выбирая subset-based алгоритм, выигрыш в точности анализа будет весомым, а потери во времени работы алгоритма незначительны, так как отдельный метод программы разумно отнести к «небольшим программам».



### 4.4.2 Нечувствительный к потоку алгоритм анализ <sup>A</sup>



Рассмотрим чувствительность алгоритма анализа указателей к потоку управления. Если алгоритм анализа чувствителен к потоку управления, он может давать более точные результаты, набор целей указателей для конкретной точки в программе [3, с. 57], но и этих наборов получается, при простейшем подходе, по одному на каждую инструкцию программы, что приводит к дополнительному потреблению памяти алгоритмом анализа. Рассмотрим подробнее, за счет чего у такого алгоритма получается более точный результат. При присваивании  $a = b$  не только генерируется информация об указании переменной  $a$  на все объекты, на которые может указывать переменная  $b$ , но и уничтожается предыдущая информация о целях переменной  $a$ , за счет этого множество целей переменной  $a$  содержит меньше неактуальной для этой



точки программы информации. Но похожего результата можно достичь и другим путем, используя SSA-форму ~~для~~ внутреннего представления.

Рассмотрим работу алгоритма нечувствительного к потоку на примере 4 (это пример 2 переведенный в SSA-форму). Так как присутствует только одно

---

#### Пример 4 Повышение точности за счет использования SSA-формы

---

**Дано:**  $x_0, x_1, \dots, x_i$  — версии одной переменной  $x$

- 1:  $a_0 = \mathbf{new} \ T()$
  - 2:  $b_0 = \mathbf{new} \ T()$
  - 3:  $c_0 = \mathbf{new} \ T()$
  - 4:  $a_1 = b_0$
  - 5:  $b_1 = c_0$
  - 6:  $c_1 = a_1$
- 

присваивание каждой переменной, легко получается следующий результат:



$$Pts(a_0) = \{O_a\}, Pts(b_0) = \{O_b\}, Pts(c_0) = \{O_c\},$$

$$Pts(a_1) = \{O_b\}, Pts(b_1) = \{O_c\}, Pts(c_1) = \{O_b\}.$$

Как было показано в разделе 2 чувствительный к потоку алгоритм анализа дает идентичный результат для конечной точки примера 2 (учитывая, что для конечной точки программы  $a = a_1, b = b_1, c = c_1$ ):


$$Pts(a) = \{O_b\}, Pts(b) = \{O_c\}, Pts(c) = \{O_b\}.$$

Получается, что чувствительный к потоку анализ не дает выигрыша в точности при использования SSA-формы, это связано с тем, что не происходит того «уничтожения предыдущей информации о целях переменной», так как неактуальная информация о целях предыдущих версий переменной не переходит в последующие версии.

В итоге получается, что чувствительный к потоку алгоритм и алгоритм анализа, работающий на SSA-форме внутреннего представления, дают одинаково точные результаты и каждый требует дополнительной памяти, либо


на хранение набора целей указателей для точек программы, либо на хранение целей указателей для всех версий переменной. Но необходимо заметить, что большинство современных компиляторов уже используют SSA-форму для внутреннего представления программы, и следовательно, при прочих равных нет смысла использовать чувствительный к потоку алгоритм анализа.

#### 4.4.3 Внутрипроцедурный алгоритм анализа



Межпроцедурные алгоритмы анализа указателей сравнительно сложны для реализации, хотя и дают более точные результаты. Тем более реализация подобного алгоритма для таких языков как Java, где каждый нестатический метод является виртуальным, выходит за рамки моей квалификационной работы. Но стоит понимать, что до вызова алгоритма анализа уже может быть проведена оптимизация открытой подстановки методов, что позволит смягчить последствия от отсутствия межпроцедурного анализа.

### 4.5 Описание алгоритма анализа для однопоточных программ



В этом разделе будет приведено описание алгоритма анализа указателей для однопоточных Java программ<sup>1</sup>. Представленный алгоритм является внутрипроцедурным алгоритмом subset-based типа нечувствительного к потоку управления, работающим с внутренним представлением в SSA-форме.

Заметим, что для однопоточной программы статические поля классов можно рассматривать как глобальные переменные.

При анализе отдельно взятого метода каждой переменной ссылочного типа ставится в соответствие множество ее целей. Для локальных переменных это множество изначально пусто. Для формальных параметров и глобальных переменных необходимо сделать консервативное предположение, что две

---

<sup>1</sup>На самом деле любая Java программа является многопоточной [TODO: cite].



любых переменных из формальных параметров и глобальных переменных могут быть синонимами. Этого можно добиться, если их множество будет содержать один специальный объект  $O_{unknown}$  искусственного типа  $Unknown$  такого, что существует расширяющее преобразование этого типа к любому другому ссылочному типу.

У любого объекта, у которого есть поля, так же хранится множество целей каждого из них. У объекта  $O_{unknown}$  существует единственное поле  $field$  типа  $Object$ , и доступ к любым полям объекта  $O_{unknown}$  транслируется в доступ к полю  $field$ . Изначально множество целей поля  $field$  объекта  $O_{unknown}$  содержит один объект —  $O_{unknown}$ .

Введем следующие обозначения

$$\begin{aligned} \text{множество целей переменной } a &: \text{VarPts}(a), \\ \text{множество целей поля } f \text{ объекта } O &: \text{ObjectFieldPts}(O, f). \end{aligned}$$

Можно определить цели поля  $f$  переменной  $a$  следующим образом

$$\bigcup_{O \in \text{VarPts}(a)} \text{ObjectFieldPts}(O, f).$$

Определим, как инструкции, описанные в разделе 4.3 изменяет множество целей переменных и полей объектов.

Присваивание  $a = b$  просто расширяет множество целей переменной  $a$

$$\text{VarPts}(a) = \text{VarPts}(a) \cup \text{VarPts}(b),$$

присваивание  $a = \mathbf{null}$  не изменяет множество целей переменной  $a$ .

Присваивание  $a = b.f$  нужно трактовать как множество присваиваний переменной  $a$  значений полей объектов, на которые может указывать переменная  $b$ :

$$\mathbf{for } O \mathbf{ in } \text{VarPts}(b): \text{VarPts}(a) = \text{VarPts}(a) \cup \text{ObjectFieldPts}(O, f).$$

Присваивание  $b.f = a$  похоже на предыдущее. Его нужно трактовать как множество присваиваний полей объектов, на которые может указывать переменная  $b$ , значению переменной  $a$ :

**for**  $O$  **in**  $\text{VarPts}(b)$ :  $\text{ObjectFieldPts}(O, f) = \text{ObjectFieldPts}(O, f) \cup \text{VarPts}(a)$ .

При преобразовании значения переменной к некоторому типу  $T$  ( $a = (T)b$ ), необходимо расширить множество целей переменной  $a$  только объектами, совместимыми по присваиванию с типом  $T$ . Для этого необходимо хранить формальный тип объекта, зная его, можно определить совместимость по присваиванию ( $\text{IsAssignable}(O, T)$ ) для некоторого объекта  $O$  и ссылочного типа  $T$  (нужно ли это описывать?). Тогда для обработки такого преобразования нужно следующее действие:

$$\text{VarPts}(a) = \text{VarPts}(a) \cup \{O : O \in \text{VarPts}(b), \text{IsAssignable}(O, T)\}.$$

В анализируемом методе инструкции создания новых объектов находятся в местах создания объектов (англ. allocation-site), число которых заведомо ограничено количеством инструкций в методе. Введем функцию

## 4.6 Модель памяти языка Java

## 4.7 Описание алгоритма анализа для многопоточных программ

## Список литературы

- [1] *Касьянов, В. Н.* Методы построения трансляторов / Касьянов, В. Н., Поттосин, И. В. — Новосибирск: Наука, 1986.
- [2] *Diwan, A.* Type-based alias analysis / A. Diwan, K. S. McKinley, J. E. B. Moss // Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. — PLDI '98. — New York, NY, USA: ACM, 1998. — Pp. 106–117. <http://doi.acm.org/10.1145/277650.277670>.
- [3] *Hind, M.* Pointer analysis: haven't we solved this problem yet? / M. Hind // PASTE. — 2001. — Pp. 54–61.
- [4] *Steensgaard, B.* Points-to analysis in almost linear time / B. Steensgaard // Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — POPL '96. — New York, NY, USA: ACM, 1996. — Pp. 32–41. <http://doi.acm.org/10.1145/237721.237727>.
- [5] *Andersen, L. O.* Program Analysis and Specialization for the C Programming Language: Ph.D. thesis / Department of Computer Science, University of Copenhagen. — 1994.
- [6] *Manson, J.* The java memory model / J. Manson, W. Pugh, S. V. Adve // Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — POPL '05. — New York, NY, USA: ACM, 2005. — Pp. 378–391. <http://doi.acm.org/10.1145/1040305.1040336>.
- [7] Efficiently computing static single assignment form and the control dependence graph: Tech. rep. / R. Cytron, J. Ferrante, B. K. Rosen et al. — Providence, RI, USA: 1991.
- [8] *Shapiro, M.* Fast and accurate flow-insensitive points-to analysis / M. Shapiro, S. Horwitz // Proceedings of the 24th ACM SIGPLAN-SIGACT symposium

on Principles of programming languages. — POPL '97. — New York, NY, USA:  
ACM, 1997. — Pp. 1–14. <http://doi.acm.org/10.1145/263699.263703>.

# План-график

TODO: обновить

Задача	Время	Срок
Ознакомление с методами оптимизирующей компиляции, изучение основных понятий (SSA-форма внутреннего представления, виды статического анализа)	2 недели	15 октября
Изучение работ по анализу указателей и синонимов для Си-подобных языков	3 недели	10 ноября
Изучение спецификации Java Virtual Machine, системы типов языка Java; изучение работ по анализу указателей и синонимов для языка Java	3 недели	30 ноября
Реализация тестовой модели и простейшего алгоритма, выполняющего анализ указателей для линейного участка программы	3 недели	20 декабря
Реализация алгоритма, выполняющего внутрипроцедурный анализ для однопоточных Java программ, в рамках тестовой модели	2 недели	15 января
Изучение спецификации Java Memory Model, преобразование алгоритма для анализа многопоточных Java программ	1 месяц	15 февраля
Интеграция созданного алгоритма в оптимизирующий Java компилятор Excelsior RVM, тестирование корректности, быстродействия, точности и других характеристик	1,5 месяца	31 марта

# Оценка за февральский отчёт

Руководитель: \_\_\_\_\_ (из 10 баллов). Подпись: \_\_\_\_\_ (Павлов П. Е.)

Преподаватель: \_\_\_\_\_