МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет Механико-математический
Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Трепаков Иван Сергеевич
(Фамилия, Имя, Отчество автора)

Тема работы: Эффективная реализация таблиц виртуальных методов в языках с поддержкой ограниченного множественного наследования

«К защите допущена»

Заведующий кафедрой,
д.ф.-м.н., профессор
Марчук А.Г /_____
(фамилия , И., О.) (подпись, МП)

«…»……20…г.

Научные руководители

к.т.н.,
зав. лаб. ИСИ СО РАН
Шелехов В.И. /_____
(фамилия , И., О.) (подпись, МП)

м.н.с. ИСИ СО РАН
Павлов П.Е. /_____
(фамилия , И., О.) (подпись, МП)

«…»……20…г.

Дата защиты: «…» ……20…г.

Новосибирск, 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Одним из основных принципов объектно-ориентированного программирования является *наследование классов*, впервые введенное в языке Simula [1]. Изначально наследование было только *одиночным*, то есть класс мог наследовать функциональность только одного класса, который назывался *суперклассом*, и иерархия классов представляла собой дерево или лес. Позже появились реализации языков с *множественным наследованием* (C++ [2], Eiffel [3], Python [4] и др.), в которых стало возможным наследовать нескольких суперклассов и представлять иерархию в виде произвольного ациклического графа. Однако поддержка множественного наследования может значительно усложнить реализацию языка [5] по сравнению с одиночным наследованием.

В связи с этим многие современные языки программирования, такие как Java [6], C# [7], Objective-C [8], Ruby [9], отказались от полноценного множественного наследования в пользу *ограниченного множественного наследования* (также известного как *интерфейсное* или *гибридное* наследование). В такой модели наследования выделяются две категории типов: *классы*, которые наследуются одиночно, и *интерфейсы* (или *протоколы*), которые наследуются множественно. При этом класс может наследовать несколько интерфейсов, и тогда говорят, что класс *реализует* соответствующие интерфейсы. В отличие от полноценных классов, у интерфейсов не может быть состояния, то есть в них нельзя объявлять поля и от них нельзя создавать объекты. Такое разделение значительно упрощает реализацию языка, и в частности, реализацию полиморфных вызовов, которые

разделяются на два вида, по аналогии с инструкциями в Java байткоде[1]: *виртуальные*, если формальный тип объекта вызова является классом, и *интерфейсные*, если формальный тип — интерфейс.

Одиночное наследование классов позволяет эффективно реализовать виртуальные вызовы с помощью *таблиц виртуальных методов*, а ограничения интерфейсов позволяют использовать более эффективные и гибкие реализации интерфейсных вызовов, чем при полноценном множественном наследовании. Одна из таких реализаций основывается на использовании *таблиц интерфейсных методов* или *интерфейсных таблиц*. При таком подходе для каждого класса создается несколько таблиц, по одной на каждый суперинтерфейс, и возникает задача минимизации размера порождаемых таблиц. Существующие решения этой задачи рассматривают интерфейсные таблицы независимо от таблиц виртуальных методов, оставляя без внимания дублирование информации между ними.

В данной работе предлагается новый алгоритм раскладки интерфейсных таблиц *внутри* таблицы виртуальных методов, минимизирующий суммарный размер таблиц без дополнительных издержек на вызовы. Измерения проводились на исследовательской виртуальной машине Excelsior RVM [11], состоящей из статического компилятора языка Java [6], и среды исполнения Java-программ [10].

[1] В Java байткоде [10] виртуальные и интерфейсные вызовы осуществляются с помощью двух различных инструкций: `invokevirtual` и `invokeinterface`, соответственно.

1. ПОСТАНОВКА ЗАДАЧИ

Зафиксируем некоторый язык программирования с ограниченным множественным наследованием, далее просто *язык*. Определим в этом языке множества классов $\mathbb{C}$ и интерфейсов $\mathbb{I}$ такие, что $\mathbb{C} \cap \mathbb{I} = \varnothing$, а также *иерархию типов* $\langle \mathbb{T}, <: \rangle$ как множество $\mathbb{T} := \mathbb{C} \cup \mathbb{I}$ с *отношением подтипа* **<:**, которое задает частичный порядок на множестве $\mathbb{T}$. Множество всех предков некоторого типа T будем обозначать $\mathbb{S}_T := \{S \in \mathbb{T} \mid T <: S\} \setminus \{T\}$. Тогда ограниченное множественное наследование определяется следующими условиями на отношение **<:**:

- $\forall C \in \mathbb{C}\ S \neq \varnothing \Rightarrow \exists! B \in S\ : \forall A \in S \quad B <: A$, где $S = \mathbb{S}_C \cap \mathbb{C}$ — задает одиночное наследование классов;
- $\forall I \in \mathbb{I} : \mathbb{S}_I \subset \mathbb{I}$ — означает, что интерфейсы не могут наследовать классы.

Методом будем называть символическую ссылку на некоторую функцию, объявленную в классе или интерфейсе. На структуру или природу этой ссылки не накладывается никаких ограничений, например, это может быть строка, состоящая из имени и сигнатуры функции.

Для описания такой типовой системы будем использовать следующий набор сущностей, описанный в листинге 1.1. В листинге 1.2 и на рис. 1.1 приведен пример иерархии, описанной в этих терминах.

Листинг 1.1: Определение структур метода, класса и интерфейса

```
type Method // e.g. type Method := String
type Type := Class(super, interfaces, methods) |
             Interface(interfaces, methods)
```

Листинг 1.2: Пример иерархии классов C, B и интерфейсов I, J, K

```
K := Interface(∅,     { "c()" })
J := Interface(∅,     { "b()" })
I := Interface({J,K}, { "c()" })
B := Class(null, {J}, { "a()" })
C := Class(B,    {I}, { "b()" })
```

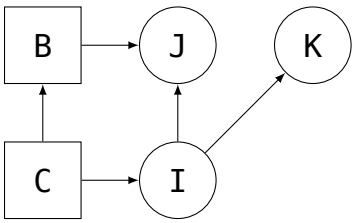


Рис. 1.1: Графическое изображение иерархии в листинге 1.2

Также определим следующие обозначения для произвольного типа T:

- $\mathbb{I}_T := \mathbb{S}_T \cap \mathbb{I}$ — множество всех суперинтерфейсов типа T;
- $\mathbb{C}_T := \mathbb{S}_T \cap \mathbb{C}$ — множество всех суперклассов типа T;
- $\mathbb{M}_T := \bigcup_{S \in \mathbb{S}_T \cup \{T\}} S.methods$ — множество методов, объявленных в типе T и в его предках.

1.1. Полиморфные вызовы

Реализацией метода будем называть непосредственно функцию, на которую ссылается этот метод. Будем считать, что реализация представлена в виде адреса, указывающего на скомпилированное тело соответствующей функции, либо на некую процедуру, которая исполняет эту функцию.

Метод можно *вызвать* от переменной формального типа, в котором этот метод объявлен. Если у метода существует несколько реализаций и статически не известно, какая из них должна быть вызвана, то такой метод называется *полиморфным*. В таком случае необходимо произвести *позднее*

связывание (англ. *late binding*) метода и его реализации во время исполнения. Вызываемая реализация определяется динамически в зависимости от типа объекта, который пришел в точку вызова, в соответствии с семантикой языка, с помощью некоторой процедуры

```
resolve: (Class, Method) → Addr,
```

которая по данному классу и методу возвращает искомую реализацию.

Листинг 1.3: Наивная реализация полиморфного вызова x.a()

```
addr := resolve(x.getClass(), "a()")
call addr
```

Далее будем считать, что все вызовы являются полиморфными и разделяются на два вида, в зависимости от формального типа объекта вызова: *виртуальные*, если формальный тип является классом, и *интерфейсные*, если формальный тип — интерфейс.

1.2. Таблица виртуальных методов

Классический способ эффективной реализации полиморфных вызовов заключается в создании для каждого класса специальной структуры *таблицы виртуальных методов* (англ. *Virtual Method Table, VMT, vtable*) — подход, впервые предложенный в языке Simula [1]. Изначально таблица виртуальных методов разрабатывалась только для случая одиночного наследования, но в последствии была адаптирована и для множественного наследования при реализации языка C++ [2], хотя это потребовало значительного усложнения структуры и дополнительных издержек на вызовы [12]. Тем не менее, благодаря своей высокой эффективности в случае одиночного наследования, таблицы виртуальных методов до сих пор успешно применяются для реализации виртуальных вызовов во многих современных

языках с ограниченным множественным наследованием [7,8,13].

1.2.1. Структура

Таблица виртуальных методов для некоторого класса C представляет собой массив vmt_C реализаций методов класса C, доступный напрямую из любого объекта этого класса. Каждому методу m класса C назначается *виртуальный номер* $vnum_C(m)$, соответствующий уникальному (в пределах класса) индексу в vmt_C, по которому доступна реализация этого метода. Таким образом виртуальный вызов генерируется в обычный косвенный вызов по статически известному индексу в таблице виртуальных методов, как показано в листинге 1.4.

Листинг 1.4: Реализация виртуального вызова x.a()

```
// vnum := vnum_C("a()") -- compile-time constant
call x.vmt[vnum]
```

Заметим, что такой косвенный вызов будет корректным, только если выполняется условие 1.1. В противном случае может вызваться некорректная реализация метода, находящаяся в ячейке с его виртуальным номером в таблице одного из наследников формального класса.

$$\forall C, B \in \mathbb{C}\ \forall m \in \mathbb{M}_B : \\ C <: B \Rightarrow vmt_C[vnum_B(m)] = resolve(C, m) \tag{1.1}$$

Обычно накладывают более сильное условие 1.2, требующее, чтобы наследники класса наследовали его виртуальные номера, что также можно переформулировать как: таблица виртуальных методов наследника должна *расширять* таблицу суперкласса. На рис. 1.2 приведен пример такого

расширения таблицы для класса C из иерархии приведенной в листинге 1.2.

$$\forall C, B \in \mathbb{C}\ \forall m \in \mathbb{M}_B :$$
$$C <: B \Rightarrow vnum_B(m) = vnum_C(m) \tag{1.2}$$

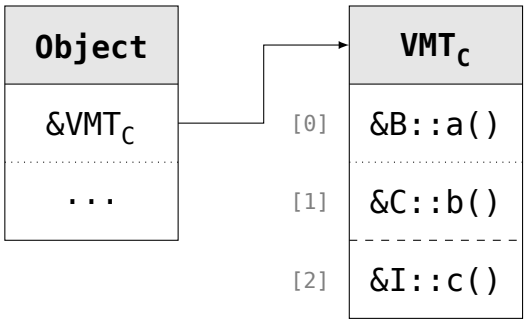


Рис. 1.2: Таблица виртуальных методов для класса C

1.2.2. Раскладка

Раскладкой таблицы виртуальных методов для класса C будем называть пару $(size_C, vnum_C)$, состоящую из размера этой таблицы и отображения виртуальных номеров класса C. Если раскладка удовлетворяет условию 1.2, то по ней можно построить корректную таблицу виртуальных методов, как показано в листинге 1.5.

Листинг 1.5: Построение VMT по раскладке для класса C

```
1 vmt_C := new Array[Addr](size_C)
2 for (m ∈ M_C) {
3   vmt_C[vnum_C(m)] := resolve(C, m)
4 }
```

В листинге 1.6 приведен алгоритм построения *базовой раскладки* таблицы виртуальных методов для класса C, которая удовлетворяет условию 1.2.

Листинг 1.6: Построение базовой раскладки для класса C

```
1 def buildVMTLayout(C) = {
2   size_C := 0
3   vnum_C := ∅
4
5   // 1. Наследование раскладки суперкласса
6   B := C.super
7   if (B ≠ null) {
8     size_C := size_B
9     vnum_C := vnum_B
10  }
11
12  // 2. Добавление новых методов класса C
13  for (m ∈ 𝕄_C if m ∉ dom(vnum_C)) {
14    vnum_C(m) := size_C
15    size_C := size_C + 1
16  }
17 }
```

1.3. Таблицы интерфейсных методов

Для реализации интерфейсных вызовов было предложено множество различных подходов, которые можно разделить на две категории: *динамические*, в которых поиск реализации производится явно в момент вызова с применением различных техник кэширования результатов, и *статические*, в которых результаты поиска вычисляются заранее до исполнения кода. Преимуществом статических подходов является константное время вызова, в то время как у динамических подходов вызовы в среднем происходят быстрее, однако в худшем случае динамический подход может оказаться даже медленнее наивной реализации. Недостатком же статических подходов является огромный размер порождаемых данных по сравнению с динамическими. Один из наиболее успешных статических подходов заклю-

чается в создании для каждой пары класса и его суперинтерфейса, *таблицы интерфейсных методов* (англ. *Interface Method Table, IMT, itable*).

1.3.1. Структура

Таблица интерфейсных методов (или *интерфейсная таблица*) для некоторого класса C и его суперинтерфейса I представляет собой массив $imt_{I,C}$ реализаций методов интерфейса I в классе C. Аналогично виртуальному случаю определяется уникальный (в пределах интерфейса) *виртуальный номер* $vnum_I(m)$ метода m в интерфейсе I, тем самым фиксируя индекс реализации в таблицах для всех классов, наследующих этот интерфейс. Таким образом, для каждой пары класс-интерфейс создается таблица, *поиск* которой делается в момент вызова, а затем делается косвенный вызов, аналогично виртуальному случаю, как показано в листинге 1.7. В простейшей реализации, используется линейный поиск по всем суперинтерфейсам класса, как показано на рис. 1.3.

Листинг 1.7: Реализация интерфейсного вызова x.a()

```
// vnum := vnum_I("a()") -- compile-time constant
imt := x.imts.find(&I)
call imt[vnum]
```

В отличие от таблицы виртуальных методов, корректность которой обеспечивалась за счет наследования виртуальных номеров, условие 1.3 выполняется автоматически за счет того, что виртуальные номера интерфейса не зависят от конкретного класса, реализующего данный интерфейс.

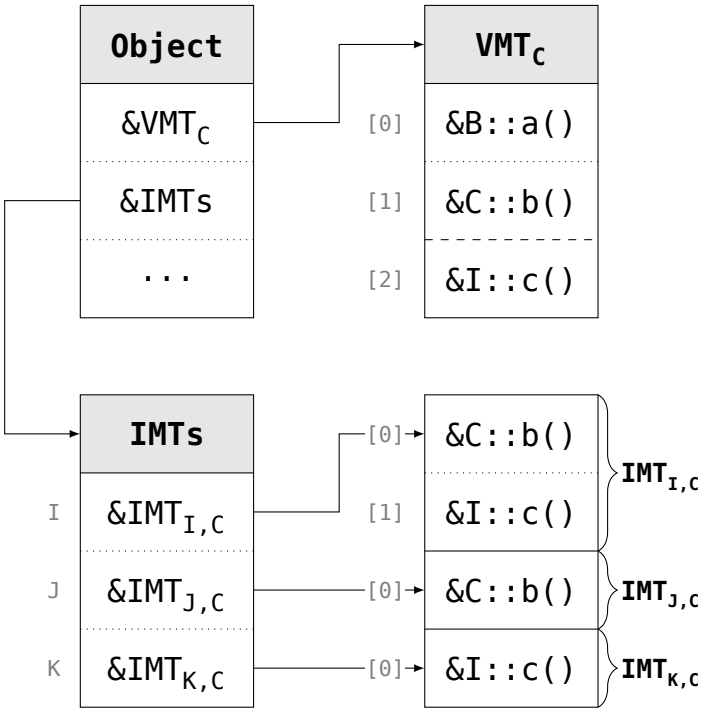


Рис. 1.3: Таблицы виртуальных и интерфейсных методов класса C

$$\forall C \in \mathbb{C}\ \forall I \in \mathbb{I}\ \forall m \in \mathbb{M}_C\ :\ C <: I\ \Rightarrow imt_{I,C}[vnum_I(m)] = resolve(C, m) \tag{1.3}$$

1.3.2. Раскладка

Раскладкой таблицы интерфейсных методов для интерфейса I будем называть пару $(size_I, vnum_I)$, состоящую из размера этой таблицы и отображения виртуальных номеров интерфейса I. Построение $imt_{I,C}$ для некоторого класса C, который реализует I представлено в листинге 1.8.

Листинг 1.8: Построение IMT по раскладке для интерфейса I в классе C

```
1 imt_I,C := new Array[Addr](size_I)
2 for (m ∈ M_I) {
3   imt_I,C[vnum_I(m)] := resolve(C, m)
4 }
```

В листинге 1.9 приведен алгоритм построения *базовой раскладки* таб-

лицы интерфейсных методов для интерфейса I.

Листинг 1.9: Построение базовой раскладки для интерфейса I

```
1 def buildIMTLayout(I) = {
2   size_I := 0
3   vnum_I := ∅
4   for (m ∈ 𝕄_I) {
5     vnum_I(m) := size_I
6     size_I := size_I + 1
7   }
8 }
```

1.3.3. Существующие модификации

У базовой реализации таблиц интерфейсных методов можно выделить два основных недостатка:

1. Линейный поиск таблицы перед каждым вызовом;
2. Большой суммарный размер таблиц.

В связи с этим, при реализации языков обычно не используют таблицы интерфейсных методов в их базовом виде, а разрабатывают собственные модификации, адресующие один или оба недостатка.

Так, например, в CACAO JVM [14] вместо линейного поиска используется поклассовая таблица соответствия интерфейсов и интерфейсных таблиц, проиндексированная в соответствии с глобальной нумерацией всех интерфейсов в программе. Такой подход позволяет реализовать интерфейсные вызовы почти так же эффективно как виртуальные, только с одной дополнительной косвенностью для получения интерфейсной таблицы. Однако на практике, большинство классов реализуют лишь несколько интерфейсов, и таблицы соответствия получаются большие и практически пустые.

Из-за этого еще больше увеличивает потребление памяти, в дополнение к размерам самих таблиц интерфейсных методов.

В Marmot [15], наоборот, поиск остается линейным, но используется модифицированная раскладка интерфейсных таблиц, в которой таблица интерфейса-наследника содержит все таблицы своих суперинтерфейсов. Благодаря такой вложенности, можно создавать только наибольшие по включению таблицы, что должно уменьшать суммарный размер таблиц. Однако существуют иерархии, на которых такое агрессивное наследование таблиц приводит не к уменьшению, а наоборот к увеличению суммарного размера таблиц.

В Jikes RVM [13] для интерфейсных вызовов применяется альтернативный подход, позволяющий не только избавиться от поиска таблиц, но и уменьшить их суммарный размер. Для каждого класса создается одна таблица фиксированного размера, при этом нескольким методам может соответствовать один виртуальный номер в этой таблице. В таком случае ячейка с этим номером содержит *процедуру разрешения конфликтов* (англ. *conflict resolution stub*), которая определяет с помощью неявного аргумента, какую из реализаций необходимо позвать. Наличие только одной таблицы позволяет превратить интерфейсный вызов в обычный косвенный, вызывающий нужную реализацию напрямую по ячейке в таблице, либо через дополнительную процедуру при наличии конфликтов. Эффективность такого подхода напрямую зависит от распределения виртуальных номеров, и в частности, от количества конфликтов. В Jikes RVM виртуальные номера назначаются в порядке загрузки классов, равномерно по фиксированному множеству индексов интерфейсной таблицы. Для такой схемы на практи-

ке можно встретить иерархии, на которых, аналогично подходу Marmot, суммарный размер данных интерфейсных таблиц (включая процедуры разрешения конфликтов) значительно превышает размер таблиц даже базовой реализации.

1.4. Задача минимизации суммарного размера таблиц

Одна из главных задач, которые приходится решать при реализации интерфейсных вызовов с помощью таблиц интерфейсных методов — это задача минимизации суммарного размера порождаемых таблиц. Поскольку для каждого класса необходимо создавать по таблице на каждый интерфейс, который он реализует, неизбежно возникает дублирование информации, то между самими таблицами интерфейсных методов, так и между таблицами виртуальных и интерфейсных методов одного класса.

Существующие решения [13,15] рассматривают интерфейсные таблицы совершенно независимо от таблиц виртуальных методов, из-за чего могут уменьшать дублирование только между самими таблицами интерфейсных методов. Также, как будет продемонстрировано в главе 3, существуют иерархии, на которых эти подходы порождают таблицы суммарно большего размера по сравнению с базовой реализацией.

Далее предлагается новый подход к реализации таблиц, использующий совмещенную раскладку таблиц виртуальных и интерфейсных методов, который позволяет сильнее ужимать размер таблиц. Более того, будет доказано, что использование такой раскладки дает результаты не хуже, чем базовая реализация таблиц.

2. СОВМЕЩЕННАЯ РАСКЛАДКА ТАБЛИЦ

Главная идея предлагаемого подхода заключается в том, чтобы расположить таблицы интерфейсных методов внутри таблицы виртуальных методов. Такое совмещение позволяет адресовать интерфейсные таблицы не по адресам в памяти, а по индексу внутри таблицы виртуальных методов, с которого таблица начинается. Эти индексы будем называть *интерфейсными номерами* и обозначать $inum_C(I)$ для каждого класса C и его суперинтерфейса I.

Введение интерфейсных номеров принципиально не меняет реализацию поиска интерфейсной таблицы, однако позволяет хранить меньше данных (вместо целого адреса таблицы хранить только индекс), и делать более эффективный косвенный вызов на некоторых архитектурах, как показано в листинге 2.1.

Листинг 2.1: Реализация интерфейсного вызова x.a() с использованием интерфейсных номеров

```
// vnum := vnum_I("a()") -- compile-time constant
inum := x.imts.find(&I)
call x.vmt[inum + vnum]
```

2.1. Базовая раскладка

Заметим, что применение интерфейсных номеров не является специфичным для предлагаемого подхода. Например, такие же интерфейсные номера можно использовать в базовой реализации, если расположить все IMT суперинтерфейсов после VMT соответствующего класса, как показано на рис. 2.1 и 2.2.

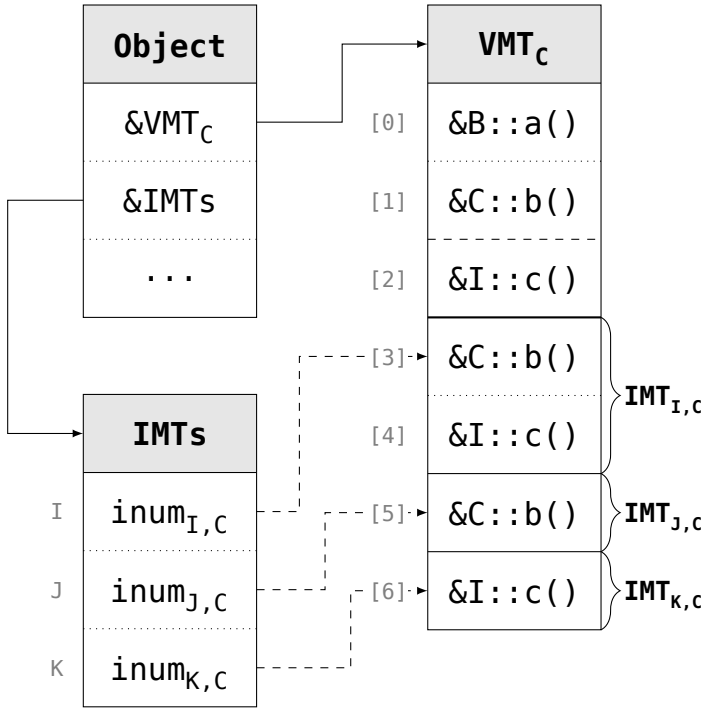


Рис. 2.1: Таблицы для класса C после введения интерфейсных номеров

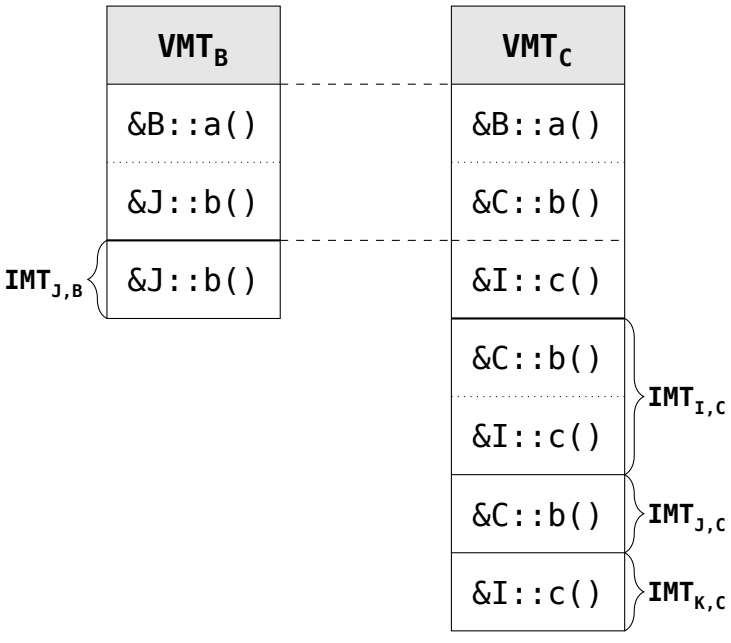


Рис. 2.2: Базовая раскладка таблиц из иерархии, изображенной на рис. 1.1

Для дальнейших оценок размеров различных раскладок, нам понадобится получить исходную оценку на суммарный размер таблиц базовой раскладки.

Размер таблицы некоторого типа T, получаемой при использовании какой-либо раскладки, зависит исключительно от выбранной раскладки, а именно от размера $size_T$. Поэтому далее будем отождествлять размер раскладки с размером таблицы полученной по этой раскладке.

Утверждение 2.1. Для базовой раскладки класса или интерфейса T выполняется равенство:

$$size_T = |\mathbb{M}_T|. \tag{2.1}$$

Доказательство. В случае, если T является интерфейсом, равенство 2.1 выполняется по построению (см. листинг 1.9). Для случая, когда T является классом, по построению (см. листинг 1.6) получаются следующие рекуррентные соотношения:

$$size_T \underset{B=T.super}{=} \begin{cases} |\mathbb{M}_T|, & B = null \\ size_B + |\mathbb{M}_T \setminus \mathbb{M}_B|, & иначе \end{cases},$$

из которых индукцией по глубине наследования выводится искомое равенство. □

Таким образом, из доказанного утверждения вытекает следующая оценка на суммарный размер таблиц базовой раскладки:

$$Size_{Base} = \sum_{C \in \mathbb{C}} (size_C + \sum_{I \in \mathbb{I}_C} size_I) \overset{\text{Утв. 2.1}}{=} \sum_{C \in \mathbb{C}} (|\mathbb{M}_C| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I|). \tag{2.2}$$

2.2. Совмещение VMT и IMT

Совмещенной раскладкой таблицы виртуальных методов класса C будем называть тройку $(size_C, vnum_C, inum_C)$, состоящую из размера таблицы, отображения виртуальных номеров методов и отображения интерфейсных номеров суперинтерфейсов класса C. При выполнении условий 1.2 и 2.3, можно построить корректную таблицу виртуальных методов, как показано в листинге 2.2.

$$\forall C \in \mathbb{C}\ \forall m_C \in \mathbb{M}_C\ \forall I \in \mathbb{I}_C\ \forall m_I \in \mathbb{M}_I : \\ vnum_C(m_C) = inum_C(I) + vnum_I(m_I) \Rightarrow m_C = m_I \tag{2.3}$$

Листинг 2.2: Построение VMT по совмещенной раскладке для класса C

```
1 vmt_C := new Array[Addr](size_C)
2 for (I ∈ 𝕀_C, m ∈ 𝕄_I) {
3   vmt_C[inum_C(I) + vnum_I(m)] := resolve(C, m)
4 }
5 for (m ∈ 𝕄_C) {
6   vmt_C[vnum_C(m)] := resolve(C, m)
7 }
```

Условие 2.3 необходимо для того, чтобы при заполнении ячеек методов класса (в строчках 5-7) не возникало конфликтов с уже записанными реализациями интерфейсных таблиц. Именно благодаря такому условию, становится возможным уменьшение размера VMT, за счет назначения методам класса виртуальных номеров, указывающих внутрь интерфейсных таблиц.

Построение совмещенной раскладки таблицы виртуальных методов, изображенной на рис. 2.3, представлено в листингах 2.3 и 2.4 и происходит

в три этапа:

1. Наследование раскладки суперкласса, включая интерфейсные номера его суперинтерфейсов, что позволяет удовлетворить условию 1.2, а также позволяет явно не добавлять в раскладку данного класса интерфейсные таблицы суперкласса;
2. Добавление *собственных* методов класса, которые не объявлены в суперклассах или суперинтерфейсах, и назначение им виртуальных номеров. В отличие от базовой раскладки из листинга 1.6, методам, унаследованным от интерфейсов, виртуальные номера будут назначены внутри интерфейсных таблиц, тем самым уменьшая дублирование между ними;
3. Добавление IMT *собственных* суперинтерфейсов, которым еще не назначен интерфейсный номер с помощью процедуры $addIMT_C$, представленной в листинге 2.4. При этом происходит назначение виртуальных номеров методам из этих таблиц, у которых виртуального номера еще нет, явно гарантируя выполнение условия 2.3.

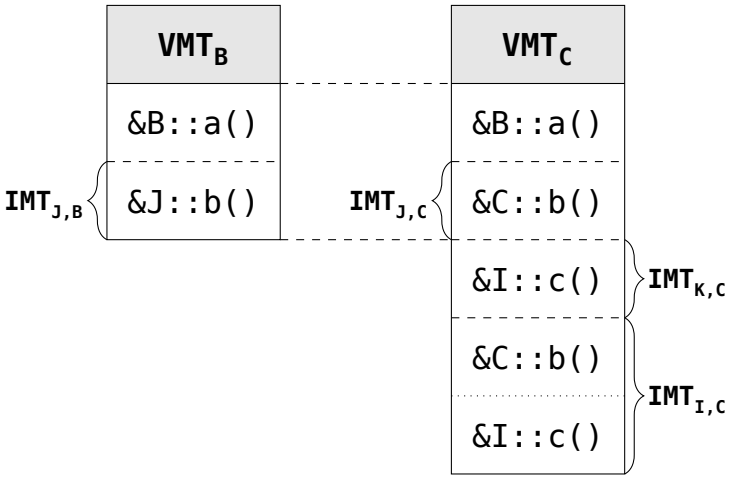


Рис. 2.3: Совмещенная раскладка таблиц из иерархии, изображенной на рис. 1.1

Листинг 2.3: Построение совмещенной раскладки для класса C

```
1 def buildVMTLayout(C) = {
2   size_C := 0
3   vnum_C := ∅
4   inum_C := ∅
5
6   // 1. Наследование раскладки суперкласса
7   B := C.super
8   if (B ≠ null) {
9     size_C := size_B
10    vnum_C := vnum_B
11    inum_C := inum_B
12  }
13
14  // 2. Добавление собственных методов класса C
15  for (m ∈ 𝕄_C if ∀S ∈ 𝕊_C : m ∉ 𝕄_S) {
16    vnum_C(m) := size_C
17    size_C := size_C + 1
18  }
19
20  // 3. Добавление IMT собственных интерфейсов
21  for (I ∈ 𝕀_C if I ∉ dom(inum_C)) {
22    addIMT_C(size_C, I)
23    size_C := size_C + size_I
24  }
25 }
```

Листинг 2.4: Процедура добавления раскладки интерфейса I в раскладку для класса C

```
1 def addIMT_C(pos, I) = {
2   inum_C(I) := pos
3
4   // Назначение виртуальных номеров внутри IMT
5   for (m ∈ 𝕄_I if m ∉ dom(vnum_C)) {
6     vnum_C(m) := inum_C(I) + vnum_I(m)
7   }
8 }
```

Заметим, что так как раскладка интерфейсных таблиц осталась базовой, то для нее верно утверждение 2.1 и выполняется равенство 2.1.

Утверждение 2.2. Для совмещенной раскладки VMT класса C выполняется:

$$size_C \le |\mathbb{M}_C| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I| \tag{2.4}$$

Доказательство. По построению раскладки (см. листинг 2.3) получаются следующие рекуррентные соотношения:

$$size_C = |M| + \begin{cases} \sum\limits_{I \in \mathbb{I}_C} |size_I|, & B = null \\ size_B + \sum\limits_{I \in \mathbb{I}_C \setminus \mathbb{I}_B} |size_I|, & иначе \end{cases},$$

где $B = C.super$ и $M = \mathbb{M}_C \setminus \bigcup\limits_{S \in \mathbb{S}_C} \mathbb{M}_S$.

Докажем утверждение 2.2 индукцией по глубине наследования классов:

База индукции ($B = null$):

$$size_C = |M| + \sum_{I \in \mathbb{I}_C} |size_I| \overset{\text{Утв. 2.1}}{=}$$
$$|M| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I| \le |\mathbb{M}_C| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I|.$$

Шаг индукции ($B \neq null$):

$$size_C = |M| + size_B + \sum_{I \in \mathbb{I}_C \setminus \mathbb{I}_B} |size_I| \overset{\text{Утв. 2.1}}{=}$$

$$|M| + size_B + \sum_{I \in \mathbb{I}_C \setminus \mathbb{I}_B} |\mathbb{M}_I| \overset{\text{По инд.}}{\leq}$$

$$|M| + |\mathbb{M}_B| + \sum_{I \in \mathbb{I}_B} |\mathbb{M}_I| + \sum_{I \in \mathbb{I}_C \setminus \mathbb{I}_B} |\mathbb{M}_I| =$$

$$|M| + |\mathbb{M}_B| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I| =$$

$$|M \cup \mathbb{M}_B| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I| \leq$$

$$|\mathbb{M}_C| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I|. \square$$

2.3. Совмещение IMT и IMT

По аналогии с таблицей виртуальных методов, можно внести таблицы суперинтерфейсов в таблицу интерфейса-наследника, как это сделано в системе Marmot [15], и определить *совмещенную раскладку* таблицы интерфейсных методов интерфейса I как тройку $(size_I, vnum_I, inum_I)$, состоящую из размера таблицы, отображения виртуальных номеров методов и отображения интерфейсных номеров суперинтерфейсов интерфейса I. Однако, как было замечено авторами Marmot, добавление всех таблиц суперинтерфейсов не всегда дает уменьшение размеров таблиц. Например, если у двух суперинтерфейсов совпадают некоторые методы, то при внесении их таблиц в таблицу интерфейса-наследника возникает дублирование информации, которого не было в базовой раскладке. Для того чтобы избежать такого дублирования, мы будем добавлять в раскладку IMT только непересекаю-

щиеся по методам таблицы суперинтерфейсов, как показано в листинге 2.5. Построение совмещенной раскладки для IMT происходит в два этапа:

1. Добавление наибольших по включению и непересекающихся по методам IMT суперинтерфейсов в порядке убывания их размеров, которым еще не назначен интерфейсный номер. При этом происходит назначение виртуальных номеров тем методам из этих таблиц, у которых виртуального номера еще нет. Добавление IMT осуществляется с помощью модифицированной процедуры $addIMT_T$ из листинга 2.4, расширенной на все типы и представленной в листинге 2.6;
2. Добавление оставшихся методов и назначение им виртуальных номеров. В отличие от базовой раскладки из листинга 1.9, методам, унаследованным от интерфейсов, виртуальные номера будут назначены внутри интерфейсных таблиц.

Листинг 2.5: Построение совмещенной раскладки для интерфейса I

```
1 def buildIMTLayout(I) = {
2   size_I := 0
3   vnum_I := ∅
4   inum_I := ∅
5
6   // 1.
       Добавление наибольших по включению и непересекающихся по методам
       IMT в порядке убывания их размеров
7   Max := {M ∈ 𝕀_I | ∀J ∈ 𝕀_I : M ∉ dom(inum_J)}
8   for (J ∈ Max ordered by (-size_J) if J ∉ dom(inum_I)) {
9     if (𝕄_J ∩ dom(vnum_I) = ∅) {
10      addIMT_I(size_I, J)
11      size_I := size_I + size_J
12    }
13  }
14
15  // 2. Добавление оставшихся методов интерфейса I
16  for (m ∈ 𝕄_I if m ∉ dom(vnum_I)) {
17    vnum_I(m) := size_I
18    size_I := size_I + 1
19  }
20 }
```

Листинг 2.6: Процедура добавления IMT интерфейса I в раскладку типа T

```
1 def addIMT_T(pos, I): Int = {
2   inum_T(I) := pos
3
4   // Назначение виртуальных номеров внутри IMT
5   for (m ∈ 𝕄_I if m ∉ dom(vnum_T)) {
6     vnum_T(m) := inum_T(I) + vnum_I(m)
7   }
8
9   // Наследование интерфейсных номеров
10  for (J ∈ dom(inum_I) if J ∉ dom(inum_T)) {
11    inum_T(J) := inum_T(I) + inum_I(J)
12  }
13 }
```

Пример таблиц, построенных по такой раскладке, изображен на рис. 2.4.

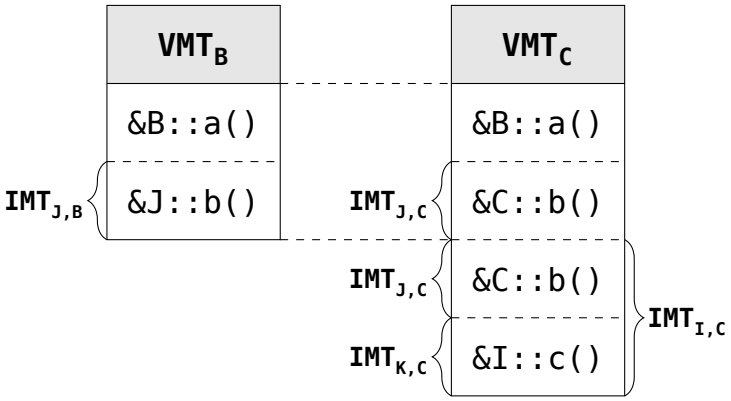


Рис. 2.4: Совмещенная раскладка таблиц из иерархии, изображенной на рис. 1.1

Таким образом, в $addIMT_T$ появляется наследование интерфейсных номеров и соответствующих подтаблиц от интерфейса, таблица которого добавляется. Заметим, что это наследование происходит как при построении IMT, так и при построении VMT, что позволяет уменьшить количество добавляемых таблиц и, как следствие, дублирование информации между ними. Для этого в совмещенной раскладке VMT из листинга 2.3 также стоит добавлять только наибольшие по включению IMT (см. листинг 2.7).

Листинг 2.7: Модификация процедуры buildVMTLayout(C)

```
19 // 3. Добавление наибольших по включению IMT
20 Max := {M ∈ 𝕀_C | ∀J ∈ 𝕀_C : M ∉ dom(inum_J)}
21 for (I ∈ Max if I ∉ dom(inum_C)) {
22   addIMT_C(size_C, I)
23   size_C := size_C + size_I
24 }
```

Заметим, что для полученной совмещенной раскладки таблиц выполняется следующее условие:

$$\forall I \in \mathbb{I}\, \forall S_1, S_2 \in dom(inum_I) : \mathbb{M}_{S_1} \cap \mathbb{M}_{S_2} = \varnothing. \tag{2.5}$$

Утверждение 2.3. Для совмещенной раскладки IMT интерфейса I вы-

полняется:

$$size_I = |\mathbb{M}_I| \tag{2.6}$$

Доказательство. По построению раскладки (см. листинг 2.5) получаются следующие рекуррентные соотношения:

$$size_I = \sum_{J \in D} |size_J| + |M|,$$

где $D = dom(inum_I)$ и $M = \mathbb{M}_I \setminus \bigcup_{J \in D} \mathbb{M}_J$.

Докажем утверждение 2.3 индукцией по количеству подтаблиц в IMT интерфейса I:

База индукции ($D = \varnothing$):

$$size_I = |M| = |\mathbb{M}_I|.$$

Шаг индукции ($D \neq \varnothing$):

$$\begin{aligned} size_C = \sum_{J \in D} |size_J| + |M| &\overset{\text{По инд.}}{\leq} \\ \sum_{J \in D} |\mathbb{M}_J| + |M| &\overset{(2.5)}{=} \\ |\bigcup_{J \in D} \mathbb{M}_J| + |M| &= \\ |\mathbb{M}_I|. \ \square & \end{aligned}$$

Следствие. Утверждение 2.2 выполняется после совмещения раскладок таблиц интерфейсных методов.

2.4. Расширение последней IMT суперкласса

Если посмотреть на структуру полученных совмещенных раскладок таблиц виртуальных и интерфейсных методов, то можно заметить, что интерфейсные таблицы суперинтерфейсов собираются в конце VMT и в начале IMT. Такая структура позволяет делать дополнительное преобразование, изображенное на рис. 2.5, позволяющее еще сильнее уменьшить размер таблиц, которое заключается в расширении последней интерфейсной таблицы суперкласса[2] с помощью интерфейсной таблицы наследника, как показано в листинге 2.8.

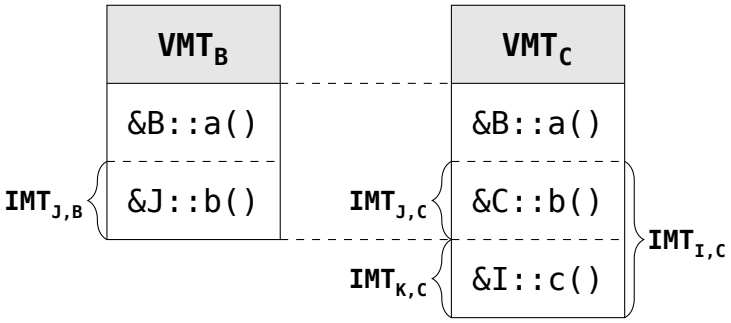


Рис. 2.5: Итоговая совмещенная раскладка таблиц из иерархии, изображенной на рис. 1.1

[2]В общем случае последних таблиц у VMT может быть несколько. Например, если класс наследует цепочку интерфейсов I **<:** J, и в интерфейсе I нет объявлений собственных методов, то последней можно считать как таблицу I, так и таблицу J. В таком случае следует перебрать все последние таблицы, начиная с наибольшей по включению (в данном примере начиная с I), и выбрать первую непустую, которую можно расширить.

Листинг 2.8: Построение итоговой совмещенной раскладки для класса C

```
1 def buildVMTLayout(C) = {
2   size_C := 0
3   vnum_C := ∅
4   inum_C := ∅
5
6   // 0. Множество наибольших по включению суперинтерфейсов
7   Max := {M ∈ 𝕀_C | ∀J ∈ 𝕀_C : M ∉ dom(inum_J)}
8
9   // 1. Наследование раскладки суперкласса
10  B := C.super
11  if (B ≠ null) {
12    size_C := size_B
13    vnum_C := vnum_B
14    inum_C := inum_B
15
16    // 1.1. Расширение последней IMT суперкласса
17    Last := {I ∈ 𝕀_B : size_I > 0 && inum_B(I) + size_I = size_B}
18    for (I ∈ Last ordered by ⟨-size_I, <:⟩;
19         J ∈ Max \ 𝕀_B if inum_J(I) = 0) {
20      addIMT_C(inum_B(I), J)
21      size_C := inum_B(I) + size_J
22      break // Расширять можно только один раз
23    }
24  }
25
26  // 2. Добавление собственных методов класса C
27  for (m ∈ 𝕄_C if ∀S ∈ 𝕊_C : m ∉ 𝕄_S) {
28    vnum_C(m) := size_C
29    size_C := size_C + 1
30  }
31
32  // 3. Добавление наибольших по включению IMT
       в порядке возрастания их размеров
33  for (I ∈ Max ordered by size_I if I ∉ dom(inum_C)) {
34    addIMT_C(size_C, I)
35    size_C := size_C + size_I
36  }
37 }
```

Для того чтобы повысить эффект от срабатывания представленного пре-

образования, применяется жадная эвристика, которая помещает наибольшую IMT последней в таблицу VMT так, чтобы в наследниках расширялась всегда наибольшая IMT. Реализована эта эвристика в виде сортировки IMT по возрастанию их размеров при добавлении в VMT.

Заметим, что оценка на размер таблиц из утверждения 2.2 остается верной, так как расширение последней IMT суперкласса может только уменьшить размер таблицы.

Таким образом, суммарный размер таблиц совмещенной раскладки не превосходит суммарного размера таблиц базовой раскладки:

$$Size_{Combined} = \sum_{C \in \mathbb{C}} size_C \overset{\text{Утв. 2.2}}{\leq} \sum_{C \in \mathbb{C}} (|\mathbb{M}_C| + \sum_{I \in \mathbb{I}_C} |\mathbb{M}_I|) \overset{(2.2)}{=} Size_{Base}. \tag{2.7}$$

3. РЕЗУЛЬТАТЫ ИЗМЕРЕНИЙ

Итоговая совмещенная раскладка была реализована в исследовательской виртуальной машине Excelsior RVM для языка Java (подробное описание реализации приведено в приложении). Изначально в виртуальной машине использовалась базовая раскладка (см. листинги 1.6 и 1.9). Тестирование проводилось на следующем наборе приложений:

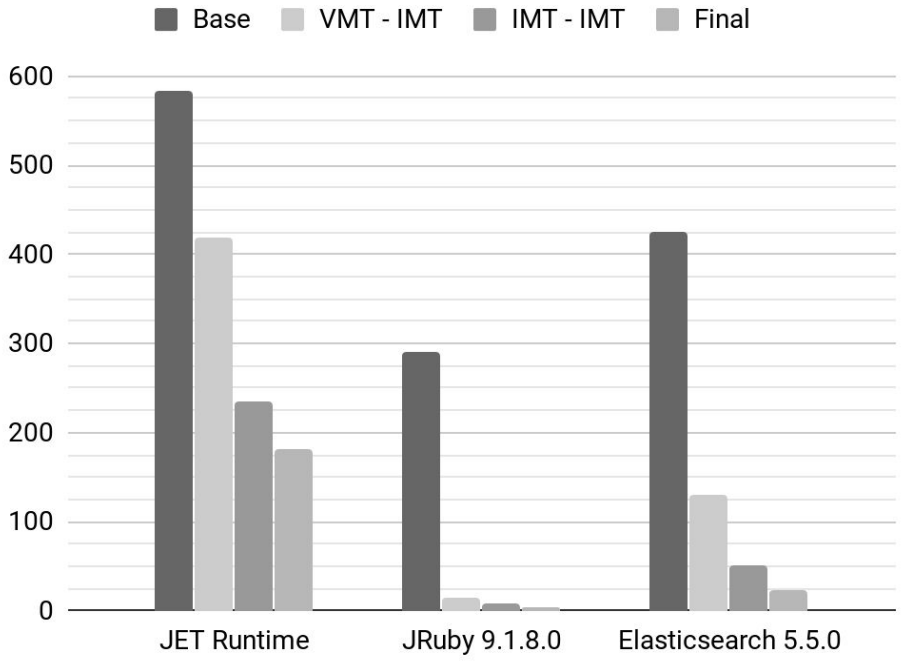
- JET Runtime — среда исполнения виртуальной машины Excelsior RVM, включающая основную часть стандартной библиотеки Java и JIT-компилятор, написанный на Java и Scala [16];
- JRuby 9.1.8.0 — реализация языка JRuby [17], написанная на Ruby [9] и Java;
- Elasticsearch 5.5.0 — популярная поисковая система [18], написанная на Java;
- Scalac 2.11.7 — самокомпилирующийся компилятор языка Scala;
- Kotlinc — самокомпилирующийся компилятор языка Kotlin [19];
- Eclipse 4.6.1 — интегрированная среда разработки Eclipse Neon, написанная на Java;
- IDEA 2017.2.1 — интегрированная среда разработки Intellij IDEA Community Edition, написанная на Java.

Данные, представленные в результатах, получены при помощи статического компилятора Excelsior RVM для архитектуры x86_64 и агрегированы по всем классам приложения, без учета динамической загрузки классов.

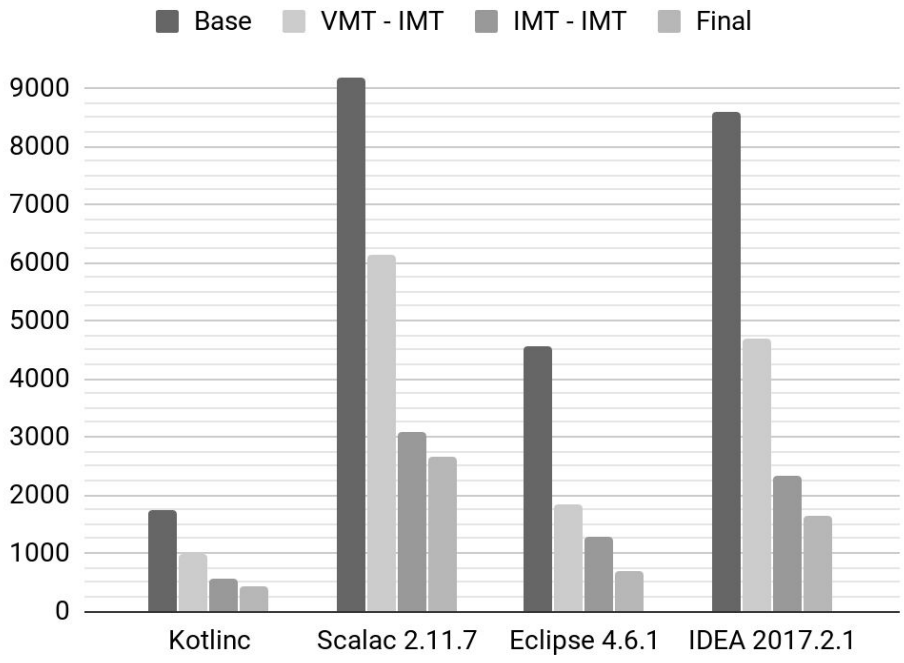
Таблица 3.1: Размеры иерархий тестируемых приложений

Приложение	Количество классов	Количество интерфейсов
JET Runtime	7992	690
JRuby 9.1.8.0	8484	360
Elasticsearch 5.5.0	15033	1189
Scalac 2.11.7	18213	1649
Kotlinc	11563	1572
Eclipse 4.6.1	65286	8142
IDEA 2017.2.1	112941	12171

На рис. 3.1 представлено суммарное уменьшение размера интерфейсных таблиц от каждого из преобразований рассмотренных в главе 2, относительно базовой раскладки таблиц. На всех тестируемых приложениях итоговая раскладка дает более чем трехкратное уменьшение: от 69% до 98% от исходного размера таблиц.



(а)



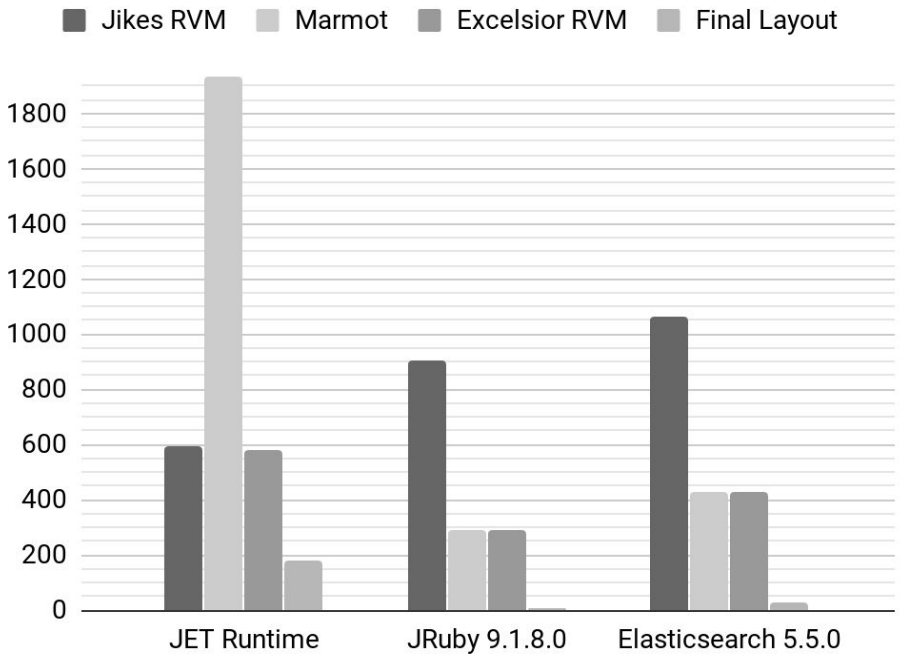
(б)

Рис. 3.1: Суммарное уменьшение размера интерфейсных таблиц, kB

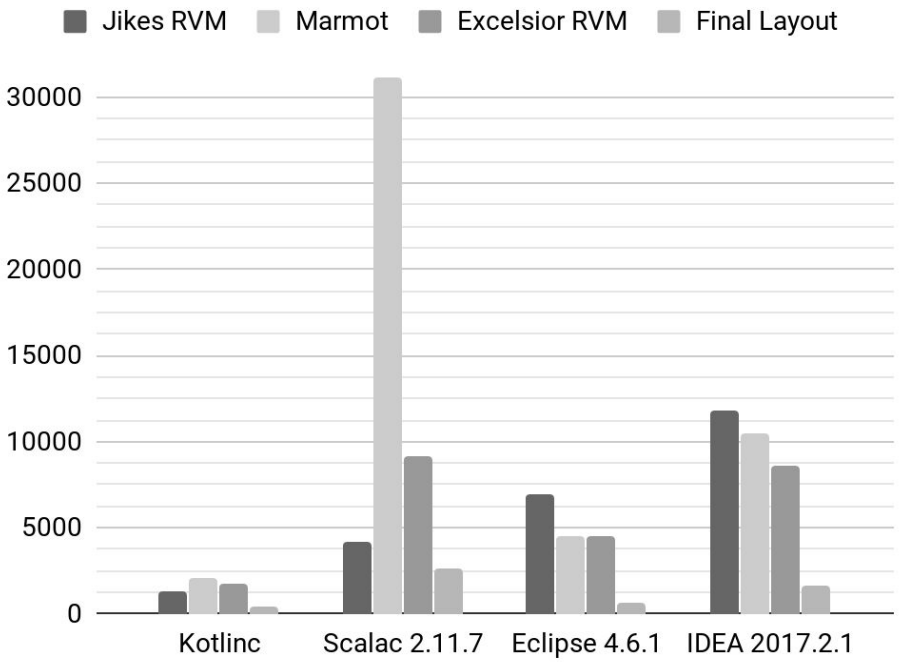
3.1. Сравнение с существующими подходами

Среди существующих реализаций таблиц интерфейсных методов, можно выделить две, которые продвинулись дальше других в решении задачи уменьшения размера порождаемых таблиц: реализации таблиц в исследовательских виртуальных машинах Jikes RVM [13] и Marmot [15], которые описаны в главе 1.3.3. Оба подхода были реализованы в Excelsior RVM[3] для проведения измерений, результаты которых представлены на рис. 3.2 вместе с исходной реализацией Excelsior RVM и итоговой раскладкой таблиц.

[3]В оригинальном подходе Jikes RVM предлагалась таблица размером в 5 элементов, и назначение виртуальных номеров делалось в порядке загрузки классов. В реализации этого подхода в Excelsior RVM используется такой же размер таблицы, а виртуальные номера назначаются в порядке обработки классов статическим компилятором.



(а)



(б)

Рис. 3.2: Суммарный размер интерфейсных таблиц в сравнении с существующими подходами, kB

4. ДАЛЬНЕЙШЕЕ РАЗВИТИЕ

В главе 2.3 представлена одна из возможных эвристик совмещения интерфейсных таблиц, которая ради неувеличения размера IMT добавляет в нее только не пересекающиеся по методам IMT суперинтерфейсов. В Marmot (см. главу 1.3.3) была предложена другая эвристика, которая добавляет все IMT суперинтерфейсов, тем самым уменьшая количество таблиц, которые необходимо создавать, но увеличивая размер самих таблиц. На самом деле оба этих подхода являются частными случаями более общей *гибридной* эвристики, выраженной следующим предикатом с параметром $p \in [0, 1]$:

$$H_p(I, J) := \begin{cases} True, & D_I = \varnothing \\ \frac{|D_I \cap \mathbb{M}_J|}{|D_I|} \leq p, & иначе \end{cases},$$

где $D_I = dom(vnum_I)$.

Гибридная эвристика определяет, таблицы каких суперинтерфейсов нужно добавлять во время построения раскладки интерфейса I, как показано в листинге 4.1. Параметр p задает отношение количества уже добавленных методов интерфейса J (которым уже назначены виртуальные номера в I) к количеству всех добавленных методов на текущий момент.

Листинг 4.1: Модификация buildIMTLayout(I) с использованием гибридной эвристики

```
// 1. Добавление наибольших по включению IMT
   в порядке убывания их размеров
Max := {M ∈ 𝕀_I | ∀J ∈ 𝕀_I : M ∉ dom(inum_J)}
for (J ∈ Max ordered by (-size_J) if J ∉ dom(inum_I) && H_p(I,J)) {
  addIMT_I(size_I, J)
  size_I := size_I + size_J
}
```

Таким образом эвристика описанная в главе 2.3 и эвристика Marmot эквивалентны гибридной эвристике с параметрами $p = 0$ и $p = 1$, соответственно. Заметим, что оба подхода могут получить результат как лучше, так и хуже другого, что продемонстрировано на рис. 4.2 и 4.3 для иерархии из листинга 4.2. Из чего следует, что перспективным направлением дальнейшей работы является изучение промежуточных значений параметра p рассмотренной гибридной эвристики, которое может улучшить полученные результаты сжатия суммарного размера таблиц.

Листинг 4.2: Пример иерархии

```
L := Interface(∅,       { "a()", "d()" })
K := Interface(∅,       { "a()", "b()", "c()" })
J := Interface({L},     { "e()" })
I := Interface({K,L},   { "a()" })
B := Class(null, {I,J}, { "a()" })
C := Class(null, {I},   ∅)
```

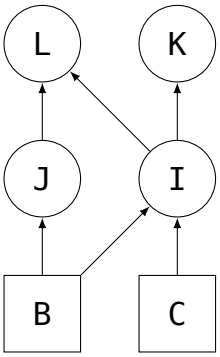
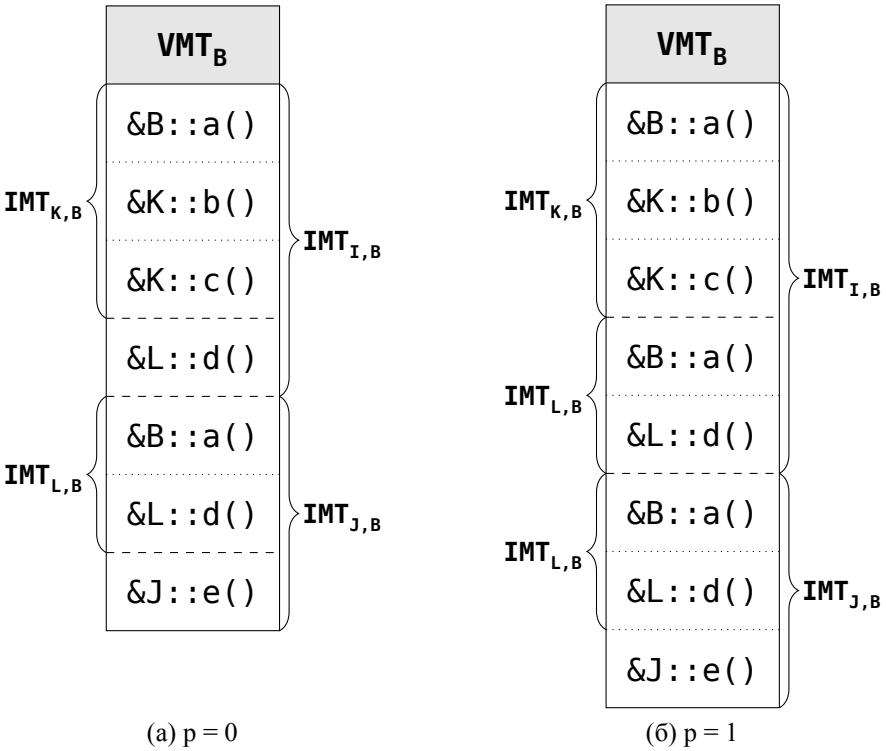


Рис. 4.1: Графическое изображение иерархии в листинге 4.2



(а) p = 0 (б) p = 1

Рис. 4.2: Таблица класса B для краевых значений гибридной эвристики

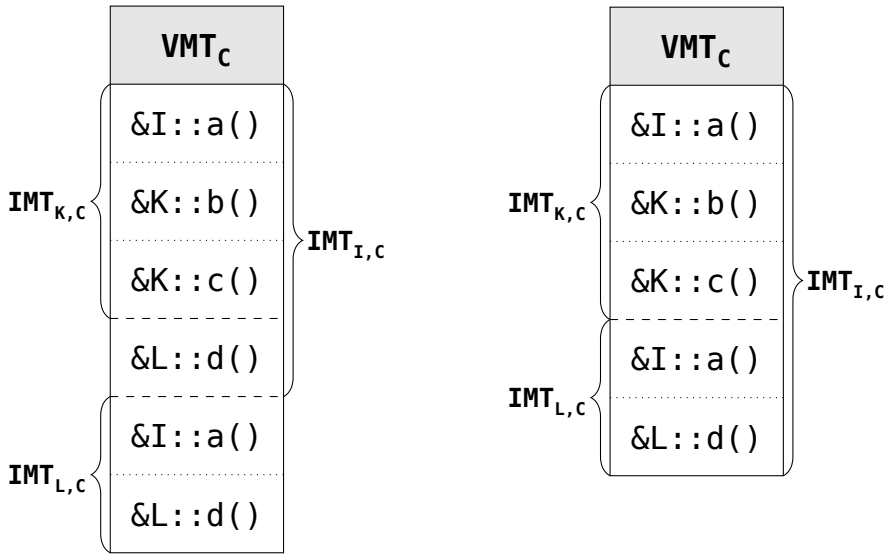


Рис. 4.3: Таблица класса C для краевых значений гибридной эвристики

ЗАКЛЮЧЕНИЕ

Основные результаты работы:

- Разработан алгоритм раскладки таблиц интерфейсных методов, для произвольного языка с ограниченным множественным наследованием;
- Доказана корректность новой раскладки и оценка суммарного размера таблиц сверху суммарным размером таблиц базовой раскладки;
- Алгоритм был реализован в виртуальной машине Excelsior RVM и апробирован на представительном наборе приложений;
- Получено значительное уменьшение суммарного размера таблиц по сравнению с базовой реализацией и с существующими решениями.

Направление дальнейших работ:

- Анализ сложности алгоритма построения раскладки и его доработка для применения в JIT компиляторе, где важна скорость построения таблиц;
- Анализ и доработка эвристик, представленных в главах 2.4 и 4;
- Разработка новых оптимизаций в Excelsior RVM, эксплуатирующих полученную вложенную структуру таблиц.

ПУБЛИКАЦИИ

1. Трепаков И. С. Эффективная реализация таблиц виртуальных методов в языках с поддержкой ограниченного множественного наследования. // Материалы 56-й Международной научной студенческой конференции МНСК-2018: Математика. Новосиб. гос. ун-т. Новосибирск, 2018.

ЛИТЕРАТУРА

1. Dahl O.-J., Myhrhaug B. SIMULA TM Implementation Guide. Norwegian Computing Center, 1973.
2. Ellis M.A., Stroustrup B. The annotated C++ reference manual. Addison-Wesley, 1990.
3. Meyer B. Eiffel: the language. Prentice-Hall, Inc., 1992.
4. Van Rossum G., Drake F.L. Python language reference manual. Network Theory, 2003.
5. Stroustrup B. Multiple inheritance for C++ // Computing Systems. 1989. Т. 2, № 4. С. 367–395.
6. Gosling J. The Java language specification. Addison-Wesley Professional, 2000.
7. Hejlsberg A., Wiltamuth S., Golde P. C# language specification. Addison-Wesley Longman Publishing Co., Inc., 2003.
8. Kochan S.G. Programming in objective-C. Addison-Wesley Professional, 2011.
9. Flanagan D., Matsumoto Y. The Ruby Programming Language: Everything You Need to Know. ” O’Reilly Media, Inc.”, 2008.
10. Lindholm T. и др. The Java virtual machine specification. Pearson Education, 2014.
11. Mikheev V. и др. Overview of excelsior JET, a high performance alternative to java virtual machines // Proceedings of the 3rd international workshop on Software and performance. ACM, 2002. С. 104–113.

12. Driesen K., Hölzle U. The direct cost of virtual function calls in C++ // ACM Sigplan Notices. ACM, 1996. Т. 31. С. 306–323.
13. Alpern B. и др. Efficient implementation of Java interfaces: Invokeinterface considered harmless // ACM SIGPLAN Notices. ACM, 2001. Т. 36. С. 108–124.
14. Krall A., Grafl R. CACAO - A 64-bit JavaVM Just-in-Time Compiler // Concurrency Practice and Experience. Chichester, Sussex: J. Wiley, c1989-c2000., 1997. Т. 9, № 11. С. 1017–1030.
15. Fitzgerald R. и др. Marmot: An optimizing compiler for Java // Software-Practice and Experience. Citeseer, 2000. Т. 30, № 3. С. 199–232.
16. Odersky M. и др. An overview of the Scala programming language. 2004.
17. Developers Jr. JRuby a Java powered Ruby implementation. Technical report, http://jruby. codehaus. org/, accessed 2008, 2008.
18. Gormley C., Tong Z. Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine. ” O’Reilly Media, Inc.”, 2015.
19. Samuel S., Bocutiu S. Programming Kotlin. Packt Publishing, 2017.
20. Йорх И.А. Эффективная интерпретация объектно-ориентированных программ в среде исполнения с ограничениями по безопасности. 2016.

ПРИЛОЖЕНИЕ

Представленный в данной работе алгоритм построения совмещенной раскладки таблиц можно применить к широкому классу языков с ограниченным множественным наследованием, за счет абстрагирования от специфики конкретного языка с помощью абстрактного понятия *метода* и процедуры `resolve`, в которой заключена логика поиска реализации для метода. В данном разделе представлены особенности реализации предложенного подхода для языка Java, а также исходный код прототипной реализации, написанной на Scala.

Везде далее термин *метод* будет употребляться в классическом смысле функции, объявленной в некотором типе. По умолчанию в Java метод может быть переопределен в наследниках класса или интерфейса, в котором этот метод объявлен. Такие методы, объявленные в классе будем называть *виртуальными*, а объявленные в интерфейсе — *интефрейсными*.

П.1. Область видимости методов в Java

По умолчанию виртуальные методы доступны или видны только из классов и интерфейсов, находящихся в том же пакете, что и объявляющий класс (будем называть такой доступ `package-private`), а интерфейсные методы являются публичными и доступны из любого типа. Как и во многих объектно-ориентированных языках, в Java можно задавать область видимости виртуальных методов. Осуществляется это с помощью модификаторов:

- `public` — *публичный* метод, доступен из любого класса или интер-

фейса;

- `protected` — метод доступен только в объявляющем классе и его наследниках;
- `private` — *приватный* метод, доступен только из класса, в котором он объявлен.

Так как приватные методы не доступны в наследниках, то для них компилятор всегда может сгенерировать прямой вызов. Остальные методы могут быть переопределены[4] и, следовательно, вызываются виртуально или интерфейсно.

Нетривиальная ситуация возникает, когда класс A и наследник B находятся в разных пакетах, и определяют `package-private` метод с одинаковой сигнатурой. В таком случае виртуальные вызовы от переменной формального типа A, в соответствии со спецификацией языка Java, должны вызывать реализацию из метода A, даже если в точку вызова пришел объект типа B, а виртуальные вызовы от переменной формального типа B должны вызывать реализацию типа B. Если в качестве ссылки на метод использовать только имя и сигнатуру, то оба метода получат одинаковый виртуальный номер, и для определения реализации придется использовать дополнительную процедуру-переходник, которая определяет реализацию метода динамически.

Альтернативно, можно избежать дополнительной косвенности, если заводить отдельные ячейки в VMT под `package-private` методы. Один из способов этого добиться, заключается в том, чтобы различать ссылки на

[4]Рассмотрение модификаторов `static` и `final` опущено для упрощения изложения.

package-private методы и на обычные методы, для того чтобы им назначались различные виртуальные номера. В реализации, которая представлена далее в листинге П.1, для такого разделения введен специальный тип ссылки PackagePrivateMethod, который отличается от обычной ссылки NormalMethod наличием имени пакета, в котором был объявлен package-private метод.

П.2. Полиморфные вызовы в Java

Для осуществления виртуальных и интерфейсных вызовов в Java байткоде используются инструкции invokevirtual и invokeinterface, которые различаются не только формальным типом переменной вызова, но еще и степенью статической и динамической верификации в соответствии со спецификацией JVM [10]: для invokevirtual гарантируется совместимость формального типа переменной вызова и настоящего типа объекта во время исполнения, но для invokeinterface такой гарантии нет, и проверка соответствия типа должна произойти во время исполнения с выбрасыванием исключения IncompatibleClassChangeError, если проверка провалилась. В реализации, использующей таблицы интерфейсных методов, эту проверку можно совместить с поиском таблицы нужного интерфейса и породить исключение, если таблица не найдена.

Помимо проверки совместимости типов, выбрасыванием исключения может закончиться и сам вызов в случаях, когда вызываемая реализация оказалась абстрактной (AbstractMethodError), недоступной из формального типа вызова (IllegalAccessError) или если вызываемых реализаций

оказалось несколько (`IllegalClassChangeError`) при множественном наследовании интерфейсов. Во всех этих случаях при построении таблицы виртуальных или интерфейсных методов вместо адреса на реализацию метода в таблицу вставляется адрес процедуры, которая выбрасывает соответствующее исключение.

Единственная проблема при реализации совмещенной раскладки возникает, если при совмещении ячеек VMT и IMT, вызываются разные реализации в зависимости от того, виртуально или интерфейсно был позван соответствующий метод. Например, в Java виртуальный метод, объявленный как `protected` в классе C, может быть вызван виртуально в любом наследнике этого класса, но при интерфейсном вызове этого метода должно выброситься исключение `IllegalAccessError`, потому что `protected` метод нельзя позвать интерфейсно[5]. В такой ситуации в конфликтующую ячейку можно вставлять адрес специальной процедуры, которая по мета-информации о вызове, доступной во время исполнения [20], определяет тип вызова и вызывает нужную реализацию.

[5]Такая ситуация может произойти при раздельной компиляции класса C и его наследника, если наследник реализующий интерфейс, был скомпилирован до того, как в классе C данный метод стал `protected`.

П.3. Реализация

В листинге П.1 представлена реализация построения совмещенной раскладки таблиц для языка Java, написанная на Scala.

Листинг П.1: Реализация совмещенной раскладки для Java

```
1  // LayoutBuilder.scala
2
3  import scala.collection.mutable
4
5  object LayoutBuilder {
6
7    trait Layout {
8      var ready = false
9      var size = 0
10     val vnum = mutable.HashMap.empty[Method, Int]
11     val inum = mutable.HashMap.empty[Interface, Int]
12   }
13
14   abstract class Type extends Layout {
15     def interfaces: Seq[Interface]
16     def methods: Seq[Method]
17   }
18
19   case class Class(superclass: Class,
20                    interfaces: Seq[Interface],
21                    methods: Seq[Method]) extends Type
22
23   case class Interface(interfaces: Seq[Interface],
24                        methods: Seq[Method]) extends Type
25
26   abstract class Method
27
28   case class PackagePrivateMethod(name: String,
29                                   sig: String,
30                                   packageName: String) extends Method
31
32   case class NormalMethod(name: String,
33                           sig: String) extends Method
34
35   def traverse(interfs: TraversableOnce[Interface]): Iterator[Interface] = {
36     interfs.toIterator.flatMap(i => Iterator(i) ++ traverse(i.interfaces))
37   }
38
39   def maxIMTs(t: Type) = {
40     val allInterfs = traverse(t.interfaces).toSeq.distinct
```

```
41      allInterfs foreach buildIMTLayout
42      val nestedIMTs = allInterfs.flatMap(_.inum.keySet).toSet
43      allInterfs filterNot nestedIMTs
44    }
45
46    def addIMT(pos: Int, t: Type, i: Interface): Unit = {
47      t.inum(i) = pos
48      t.size = pos + i.size
49
50      for ((m, k) <- i.vnum if !t.vnum.isDefinedAt(m)) {
51        t.vnum(m) = pos + k
52      }
53
54      for ((j, k) <- i.inum if !t.inum.isDefinedAt(j)) {
55        t.inum(j) = pos + k
56      }
57    }
58
59    def buildVMTLayout(c: Class): Unit = {
60      if (c.ready) return
61
62      val interfs = maxIMTs(c)
63
64      // extend superclass
65      if (c.superclass != null) {
66        buildVMTLayout(c.superclass)
67        c.size += c.superclass.size
68        c.vnum ++= c.superclass.vnum
69        c.inum ++= c.superclass.inum
70
71        val lastIMTs = c.inum.keySet.filter(i => i.size > 0 &&
72                                                 c.inum(i) + i.size == c.size)
73
74        def extendingOrder(x: Interface, y: Interface): Boolean = {
75          if (x.size == y.size) x.inum.isDefinedAt(y)
76          else x.size > y.size
77        }
78
79        var extended = false
80        for {
81          last <- lastIMTs.toSeq.sortWith(extendingOrder) if !extended
82          extending <- interfs.find(i => !c.inum.isDefinedAt(i) &&
83                                         i.inum.isDefinedAt(last) &&
84                                         i.inum(last) == 0)
85        } {
86          addIMT(c.inum(last), c, extending)
87          extended = true
88        }
89      }
90
```

```
91      val inherited = c.vnum.keySet | interfs.flatMap(_.vnum.keySet).toSet
92      // add new declared methods
93      for (m <- c.methods if !inherited(m)) {
94        c.vnum(m) = c.size
95        c.size += 1
96      }
97
98      // add maximal IMTs
99      for (i <- interfs.sortBy(_.size) if !c.inum.isDefinedAt(i)) {
100       addIMT(c.size, c, i)
101     }
102
103     c.ready = true
104   }
105
106   def buildIMTLayout(i: Interface): Unit = {
107     if (i.ready) return
108
109     def isDisjoint(s: Interface) = (s.vnum.keySet & i.vnum.keySet).isEmpty
110     // add disjoint IMTs
111     for (s <- maxIMTs(i).sortBy(-_.size) if isDisjoint(s)) {
112       addIMT(i.size, i, s)
113     }
114
115     // add new declared methods
116     for (m <- i.methods if !i.vnum.isDefinedAt(m)) {
117       i.vnum(m) = i.size
118       i.size += 1
119     }
120
121     i.ready = true
122   }
123 }
```