УДК: 004.4'422

Название: Построение блоков обработки исключений при декомпиляции Java-байткода

Авторы:

Соловьев В.В. (Институт систем информатики им. А.П. Ершова СО РАН, Новосибирский государственный университет),

Липский Н.В. (Институт систем информатики им. А.П. Ершова СО РАН)

Аннотация: Задача декомпиляции Java-байткода состоит в построении исходного кода на языке Java, эквивалентного данному байткоду. Байткод — линейная программа с условными и безусловными переходами, а язык Java содержит структуры управления, который образуют иерархию в исходном коде. Эту иерархию необходимо восстанавливать при декомпиляции, в частности, необходимо восстановить блоки обработки исключений try-catch-finally. В проекте Excelsior RVM (виртуальной машины Java со статическим компилятором) байткод декомпилируется для проведения оптимизирующих преобразований. При построении блоков обработки исключений декомпилятор системы Excelsior RVM полагает, что байткод был получен путем компиляции исходного кода стандартным компилятором языка Java, и пытается совершить обратное преобразование. Иногда это не удается для байткода, полученного другими инструментами. В данной работе предложен алгоритм декомпиляции, восстанавливающий блоки обработки исключений из произвольного корректного байткода. Этот алгоритм реализован, интегрирован в систему Excelsior RVM и протестирован на реальных приложениях.

Ключевые слова: компиляция, декомпиляция, язык програмирования Java, Java-байткод, обработка исключений, блоки обработки исключений, таблицы защищенных интервалов

1. Введение. Исходный текст на языке Java [14] переводится Java-компилятором в двоичный формат (Java-байткод) виртуальной машины Java (JVM) [9]. Java-байткод является корректным, если он проходит процедуру верификации [9]. Рассматривается задача декомпиляции корректного байткода, а точнее — подзадача построения блоков обработки исключений. В декомпиляторе системы Excelsior RVM восстановление блоков обработки исключений предваряет восстановление прочих типов структур управления [1]. Таким образом, подзадачу построения блоков обработки исключений можно рассматривать независимо от прочих подзадач декомпиляции.

Блок обработки исключений языка Java выглядит следующим образом:

```
try { try-блок }
catch (CatchType1 e1) { catch-блок 1 }
```

```
…
catch (CatchType_N e_N) { catch-блок N }
finally { finally-блок }
```

Два блока обработки исключений не могут иметь общих catch-блоков и не могут текстуально пересекаться друг с другом кроме строгого вложения.

Таблица защищенных интервалов Java-байткода — это массив структур типа *интервал*. Каждый интервал — это четверка <StartPC, EndPC, HandlerPC, CatchType>, где StartPC, EndPC и HandlerPC — адреса инструкций в байткоде, а CatchType — идентификатор ссылочного типа. Для выражения отношения наследования будем применять символ <:, если X — наследник Y или равен Y, то будем записывать это как X <: Y. Семантика обработки исключений такова: если во время исполнения инструкции с адресом PC возникает исключение типа Type, то в таблице защищенных интервалов ищется первый интервал, удовлетворяющий свойствам: StartPC ≤ PC, PC < EndPC, CatchType <: Type. Если такой интервал находится, то управление переходит на инструкцию с адресом HandlerPC. Иначе исключение передается в метод, вызвавший данный. В корректном байткоде интервалы должны удовлетворять условиям: StartPC, EndPC, HandlerPC — корректные адреса инструкций; StartPC < EndPC; CatchType <: Throwable.

Стандартный компилятор языка Java транслирует блоки обработки исключений в таблицу защищенных интервалов по следующему алгоритму:

1) Try-блок компилируется в непрерывный участок байткода [SPC, EPC).
2) Catch-блок$_i$ компилируется в байткод, начинающийся с инструкции HPC$_i$.
3) Finally-блок компилируется по-разному в зависимости от версии компилятора
    a. Для версии Sun JDK 1.4 и более ранних finally-блок компилируется в байткод, который вызывается через инструкции jsr/ret из каждой точки выхода всех try, catch и finally блоков.
    b. В более поздних версиях компилятор копирует finally-блок и подставляет его в каждую точку выхода, выполняя, по сути, открытую подстановку подпрограммы, ранее реализуемую через jsr/ret.
4) В таблицу защищенных интервалов вставляются записи <SPC, EPC, HPC$_i$, CatchType$_i$> для каждого catch-блока. Также вставляются записи, покрывающие скомпилированный try-блок и все catch-блоки с HandlerPC, указывающим на finally-блок и CatchType, равным Throwable, для того чтобы finally-блок выполнялся в случае любого выхода.
5) Записи вносятся в таблицу в последовательности топологической сортировки по отношению текстуального вложения. Прежде чем будут внесены записи,

соответствующие некоторому блоку обработки исключений, в таблице уже должны быть записи, соответствующие всем блокам, вложенным в него.

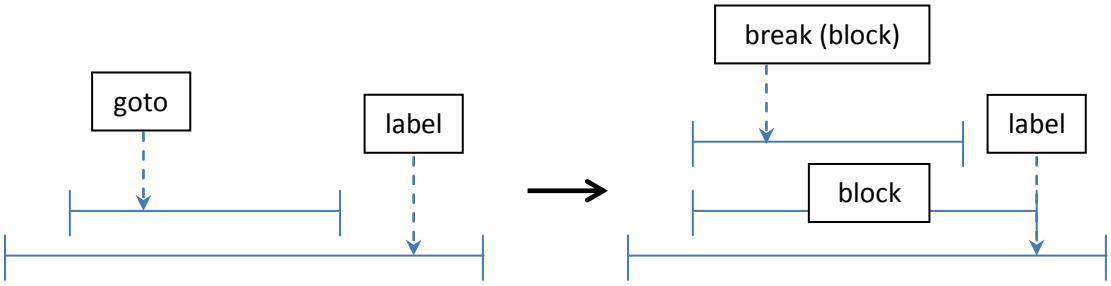
Предполагая, что таблица и байткод были получены стандартным компилятором языка Java, можно предложить *алгоритм прямой декомпиляции*:

1) try-блок определяется по StartPC и EndPC. Все записи таблицы, у которых совпадают эти границы, объединяются в одну группу;
2) для группы записей строится набор catch-блоков (все блоки, доминированные блоком, начинающимся с HandlerPC, попадают в catch-блок). Для этого строится Control Flow Graph [5];
3) в зависимости от того, как компилировался finally-блок, либо выбирается байткод, заключенный между jsr/ret инструкциями, либо ищется скопированный байткод, соответствующий finally-блоку;
4) выбранные группы байткода декомпилируются, и результат заключается в блок обработки исключений. CatchType_i берутся из таблицы интервалов;

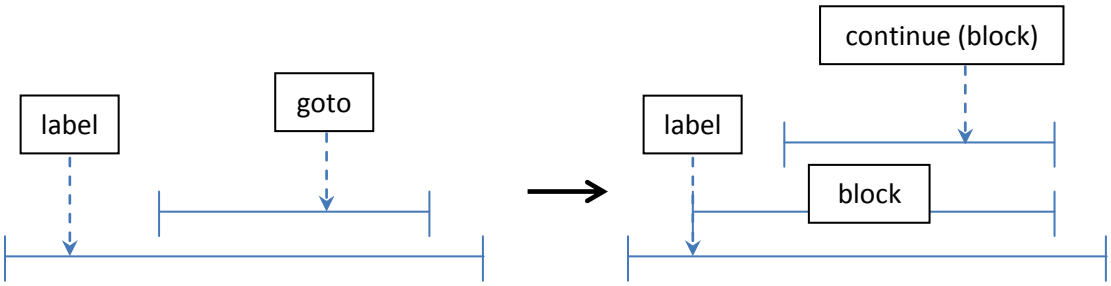
Произвольный байткод может быть получен с помощью обфускаторов, инструментации байткода, автоматической модификации или генерации кода (например, с помощью аспектно-ориентированного программирования), трансляцией JVM-based языков (JRuby, Jython, Scala) или программированием на Java ассемблере. Таблицы защищенных интервалов в таком байткоде могут не подходить для преобразования, обратного к преобразованиям, совершаемым стандартным компилятором языка Java. При постановке задачи декомпиляции произвольного корректного байткода можно полагаться только на ограничения, накладываемые верификатором. В частности, интервалы могут пересекаться без вложения одного в другого, у разных интервалов могут быть общие HandlerPC и т.п. Такие таблицы не могут быть декомпилированы прямым алгоритмом.

Результатом работы декомпилятора системы Excelsior RVM является дерево абстрактного синтаксиса (AST) [11], из которого можно получить исходный код на языке Java. В алгоритмах построения AST будет использоваться оператор goto. Чтобы представление оставалось структурным (возможность построения из него исходного кода), операторы goto должны моделироваться операторами break, continue и block языка Java. *Структурным оператором goto* будем называть оператор goto, который переводит исполнение на оператор label, если оператор label находится в произвольном месте цепочки, в которой находится или в которую вложен оператор goto (вложение может быть через несколько уровней). Структурные операторы goto можно смоделировать операторами break, continue и block. Это продемонстрировано на рис. 1. В части a) показано, как моделируется структурный оператор goto операторами break и block, в части

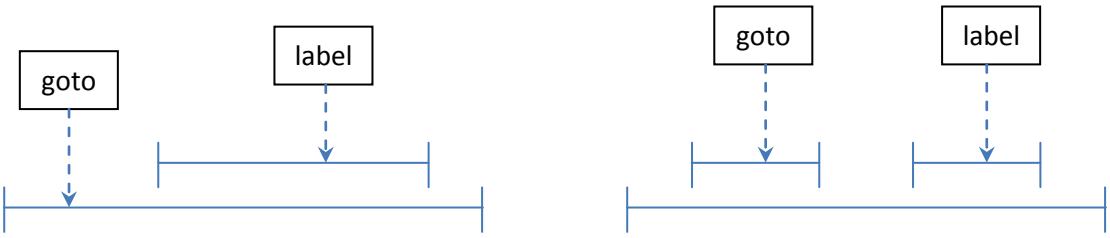
б) показано, как другой структурный оператор goto можно смоделировать операторами continue и block, в части в) показаны два примера неструктурного оператора goto.



а) Моделирование структурного оператора goto с помощью block и break



б) Моделирование структурного оператора goto с помощью block и continue



в) Примеры неструктурных переходов goto

Рис. 1

2. Формальная модель семантики обработки исключений. Разработанные нами алгоритмы декомпиляции блоков обработки исключений преобразовывают таблицы защищенных интервалов, поэтому нам необходимо доказать, что семантика обработки исключений сохраняется после преобразований. Для этого формализуем это понятие и введем ряд эквивалентных преобразований.

2.1. Определения. *Множество инструкций байткода* PC = 0, ..., L. PC $\subset$ N (натуральные числа). *Множество типов исключений* <XTypes, <:>, <: — частичный порядок, означающий наследование. Наименьший элемент XTypes — Throwable. *Множество интервалов* XIntervals = {<S, E, T, H>}, S, E, H $\in$ PC, S < E, T $\in$ XTypes.

Говоря об интервале A, равном <S, E, T, H>, будем обозначать S, E, T и H, как A.Start, A.End, A.Type и A.Handler соответственно. Целочисленный интервал [A.Start, A.End) будем обозначать A.Range. *Множество таблиц исключений* XTables = {{Int_i | Int_i ∈ XIntervals, i = 0..k}} — множество всех конечных упорядоченных последовательностей элементов из XIntervals. Говоря о таблице X ∈ XTables, X = (Int_1, …, Int_k), будем обозначать Int_i, как X_i, а k как |X|. *Предикат риска* RF: PC → {0, 1}. RF(X) = 1 ⇔ инструкция с адресом X может вызывать исключение. *Безопасный код* NRF = {c ∈ PC, RF(c) = 0}. *Предикат безопасности* UTF: XIntervals → {0, 1}. UTF(X) = 1 ⇔ X.Range ⊂ NRF. *Функция интерпретации*: Δ: XTables x (PC / NRF) x XTypes → PC ∪ {⊥}

Δ(<XT, C, Type>) =

1)XT_i.Handler, где i = min{i | C ∈ XT_i.Range и Type <: XT_i.Type};

2)⊥, если такого i нет.

Отношение эквивалентности на XTables: XT_1 ≈ XT_2 ⇔ ∀ C ∈ PC/NRF, ∀ type ∈ XTypes: Δ(XT_1,C,type) = Δ(XT_2,C,type). Преобразование Ψ: XTables → XTables будем называть *эквивалентным преобразованием*, если ∀ XT ∈ XTables: XT ≈ Ψ (XT).

2.2. Система эквивалентных преобразований. Определим три преобразования. Первое — Sep, разбивает некоторый интервал таблицы по некоторой инструкции и вставляет два подинтервала в таблицу вместо исходного. Второе — Del, удаляет интервал из таблицы, если байткод, ограниченный этим интервалом, не может бросать исключения. Третье — Concat, объединяет два подряд идущих интервала в таблице, если между ними находится код, который не может бросать исключения. Интуитивно понятно, что эти преобразования сохраняют семантику обработки исключений, поэтому формальное доказательство этого факта будет приведено в приложении A, а в дальнейшей работе будем полагаться на его справедливость.

Пусть XT ∈ XTables. Преобразование $Sep_{n,c}$(XT), где c ∈ (XT_n.Start, XT_n.End) – *разбиение n-ого интервала таблицы по инструкции c*, по следующему правилу:

$Sep_{n,c}$ (XT) = SepT такая, что $SepT_i$ =

1)XT_i, если i < n

2)<XT_n.Start, C, XT_n.Type, XT_n.Handler>, если i = n

3)< C, XT_n.End, XT_n.Type, XT_n.Handler>, если i = n + 1

4)XT_{i+1}, если i > n + 1

Преобразование $Del_n(XT)$, где $UTF(XT_n) = 1$ — *удаление безопасного интервала n*, по следующему правилу:

$Del_n(XT)$ = DelT такая, что $DelT_i$ =

1)XT_i, если $i < n$

2)XT_{i+1}, иначе

Преобразование $Concat_n(XT)$, где $XT_n.End < XT_{n+1}.Start$, $XT_n.Type = XT_{n+1}.Type$, $XT_n.Handler = XT_{n+1}.Handler$, $[XT_n.End, XT_{n+1}.Start) \subset NRF$ – *слияние n-ого и n+1-ого безопасно разделенных интервалов*, по следующему правилу:

$Concat_n(XT)$ = ConcatT такая, что $ConcatT_i$ =

1)XT_i, если $i < n$

2)$<XT_n.Start, XT_{n+1}.End, XT_n.Type, XT_n.Handler>$, если $i = n$

3)XT_{i+1}, если $i > n$

Утверждение 1: Преобразования $Sep_{n,c}$, Del_n, $Concat_n$ являются эквивалентными.

3. Алгоритм декомпиляции произвольной таблицы защищенных интервалов. Алгоритм прямой декомпиляции хорошо работает с таблицами защищенных интервалов, полученными компиляцией исходного текста Java-компилятором. В основе его работы лежит предположение, что таблица была получена прямой компиляцией, поэтому он накладывает на нее множество условий, которые обеспечивают полное восстановление блоков обработки исключений. В произвольной корректной таблице эти условия могут не выполняться, и поэтому возникают проблемы прямой декомпиляции.

3.1. Декомпиляция try-блоков. В произвольной корректной таблице два интервала могут пересекаться без вложения одного в другой (Рис. 2а). Такая таблица не может быть получена в результате прямой компиляции исходного текста на языке Java и, соответственно, не может быть напрямую декомпилирована. Также, если в исходном тексте один блок обработки исключений был вложен в другой, то при компиляции в байткод, интервалы вложенного блока будут находиться в таблице перед интервалами объемлющего. Таким образом, при выбросе исключения виртуальная машина сначала проверит интервалы вложенного блока, затем объемлющего. В то же время в верификаторе нет такого требования, и объемлющий интервал может находиться в таблице раньше, чем вложенный (Рис. 2б). Если декомпилировать оба таких интервала в блоки обработки исключений, то их вложенность будет нарушать семантику обработки исключений в исходном байткоде.

Далее будем обозначать таблицу исключений текущего декомпилируемого метода как XT. XT ∈ XTables. XT_i *пересекается без вложения* с XT_j, если XT_i и XT_j пересекаются, но ни один не вкладывается в другой. XT_i *неправильно вкладывается* в XT_j, если XT_i строго вкладывается в XT_j, и при этом i > j. Таблицу исключений будем называть *регулярной*, если в ней нет пересечений без вложения и неправильных вложений.

а) Пересечение без вложения б) Неправильное вложение

Рис. 2

3.1.1. Алгоритм проверки таблицы на регулярность. Алгоритм, проверяющий для произвольной таблицы, является ли она регулярной, на псевдоязыке высокого уровня выглядит следующим образом:

```
Stack<ExcepInfo> stack;      -- стек вложенности
Sort(XT);                    -- сортируем таблицу (по увеличению Start и, при
                             -- равных Start, по уменьшению End)
stack.push(XT[0]);           -- кладем на стек первый интервал
FOR i := 1 TO Len(XT)-1 DO
    WHILE ((stack.size > 0) AND (XT[i]  не пересекается со stack.top)) DO
        stack.pop;        -- удаляем, пока не встретим пересечение интервалов
    END;
    IF ( (stack.size > 0) AND
           ( (XT[i] пересекается без вложения со stack.top) OR
             (XT[i] неправильно вложен в stack.top))) THEN
        RETURN FALSE;         -- ситуация, когда stack.top строго вложен в
                              -- XT[i] исключена способом сортировки
    END;
    stack.push(XT[i]);        -- новая верхушка стека
END;
RETURN TRUE;
```

Утверждение 2: Алгоритм проверки таблицы на регулярность определяет, является ли таблица XT регулярной и обладает ли следующими характеристиками: трудоемкость — O(|XT| * ln(|XT|)), емкостная сложность — O(|XT|). Доказательство в приложении А.

3.1.2. Алгоритм приведения таблицы к регулярному виду. Мы собрали статистику на тестовом наборе системы Excelsior RVM и изучили примеры байткода, таблицы исключений в которых были нерегулярными. Обнаружилось, что такие таблицы

появляются в основном в обфусцированных программах либо в программах, где finally-блок был скомпилирован с применением jsr/ret инструкций. Когда finally-блок компилируется с применением jsr/ret инструкций, то инструкция jsr, вставленная внутрь try-блока, с точки зрения компилятора Java не должна включаться в защищенный интервал. Поэтому оригинальный try-блок разрезается этими инструкциями на несколько записей в таблице исключений. При этом может появиться неправильная вложенность. Еще одной частой причиной нерегулярности являются интервалы, такие что HandlerPC = StartPC и область кода, которую они защищают, неспособна создать исключение. Их удаление из таблицы исключений не влияет на семантику обработки.

Алгоритм приведения таблицы к регулярному виду

1) Применим преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1 и XT[i].Handler = XT[i].Start;
2) Применим преобразование $Concat_i(XT)$ для всех подходящих i.

Утверждение 3: Алгоритм приведения таблицы к регулярному виду сохраняет семантику обработки исключений, что следует из утверждения 1. Его временная сложность составляет O(|XT|), так как оба его шага осуществляются линейным проходом по таблице исключений, а каждое преобразование выполняется за O(1). Емкостная сложность алгоритма составляет O(1).

3.1.3. Алгоритм уравнивания интервалов. После реализации методов преобразования таблиц, мы повторно собрали статистику и обнаружили, что во всем тестовом наборе остался только один метод, таблица исключений в котором остается нерегулярной. Кроме поставленной практической задачи, которую можно считать разрешенной, была поставлена и теоретическая задача о декомпиляции произвольной таблицы исключений. Поэтому для покрытия всех случаев, нами был разработан алгоритм преобразования, работающий в тех случаях, когда таблица не является регулярной и не приводится к регулярному виду предыдущим алгоритмом.

Алгоритм уравнивания интервалов

1. Все пересечения разбиваются преобразованиями $Sep_{i,c}(XT)$ где C – границы пересекающихся интервалов.
2. Применим преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1, так как в результате первого пункта могут возникнуть такие интервалы.

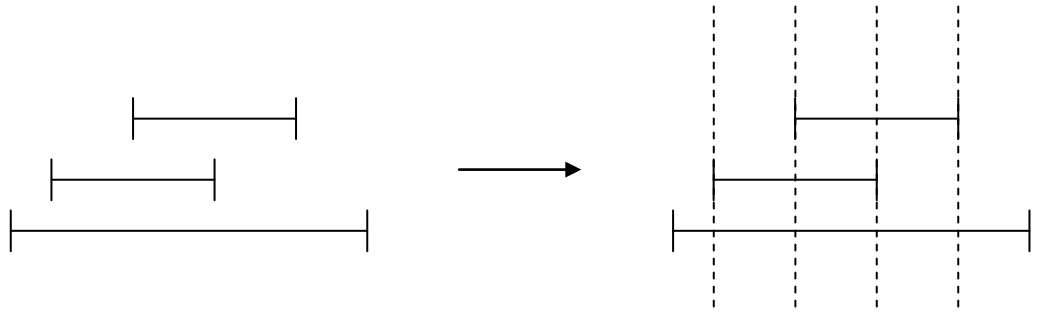


Рис. 3

Утверждение 4: Алгоритм уравнивания интервалов сохраняет семантику обработки исключений, что следует из утверждения 1. Таблица исключений после такого преобразования станет регулярной, так как все интервалы в преобразованной таблице будут либо совпадать, либо не пересекаться вообще. Каждый интервал можно порезать максимум 2*(|XT|-1) точками, причем множество этих точек можно построить заранее. Используя линейный список для хранения получаемой в процессе алгоритма таблицы, сделаем максимум |XT| *2*(|XT| -1) элементарных разбиений и |XT| *(2*|XT| + 1) операций вставки в линейный список. Таким образом, временная сложность O(|XT| *|XT|). Емкостная сложность также равна O(|XT| *|XT|).

3.2. Декомпиляция catch-блоков. В блоках обработки исключений управление в catch-блок может перейти только из соответствующего ему try-блока. В тоже время в таблице защищенных интервалов два разных интервала могут указывать на один и тот же HandlerPC, что делает невозможным прямую декомпиляцию такого байткода. Также HandlerPC может указывать в середину другого try-блока или другого catch-блока, что также невозможно представить в исходном тексте и в структурном представлении программы.

Основной идеей описываемого алгоритма декомпиляции является отказ от декомпиляции catch-блоков в смысле полного восстановления их в блок обработки исключений как цепочек операторов. Вместо этого во внутреннем представлении создается следующая структура:

```
try { try-блок }
catch (CatchType1 e1) { goto handler1; }
…
catch (CatchTypeN eN) { goto handlerN; }
```

$Handler_i$ — метка, соответствующая началу декомпилированных операторов, начинающихся с HandlerPC i-ой записи таблицы. При такой декомпиляции исчезают проблемы с совпадающими catch-блоками, принадлежащими разным интервалам. При этом построении нам нужно проверять, что операторы goto Handler являются

структурными операторами goto. XT[i].Handler является *структурным*, если для любого XT[j], такого что XT[i].Handler лежит в интервале (XT[j].Start, XT[j].End), XT[i] строго вложен в XT[j]. Нам необходимо, чтобы все Handler в таблице исключений были структурными. Проверка этого условия осуществляется за O(|XT|*|XT|).

Алгоритм разбиения интервалов

1. Проходим по таблице и проверяем для каждой пары интервалов X и Y, где X.Handler попадает внутрь Y, является ли X строго вложенным в Y. Если не является, то применяем преобразование $Sep_{i,X.Handler}(XT)$, где i – номер Y.
2. Применяем преобразование $Del_i(XT)$ для всех i таких, что UTF(XT[i]) = 1, так как в результате первого пункта могут возникнуть такие интервалы.

Утверждение 5: Временная и емкостная сложности данного алгоритма такие же, как и у алгоритма уравнивания интервалов — O(|XT| * |XT|). Алгоритм разбиения интервалов сохраняет семантику обработки исключений, что следует из утверждения 1. Этот алгоритм сохраняет регулярность таблицы, так как разбиение каждой новой границей (HandlerPC интервала X) разбивает все интервалы (кроме тех, в которые X строго вложен), и пересечений и неправильных вложений не появляется (рис. 4). Этот алгоритм увеличивает таблицу. В худшем случае (рис. 4), таблица увеличивается в 2*|XT| раз. Однако даже в таком теоретическом случае количество блоков обработки исключений останется 2*|XT|.

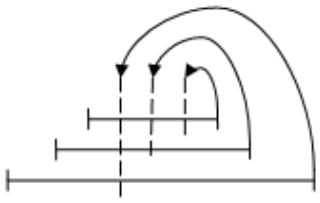


Рис. 4

3.3. Декомпиляция finally-блоков. Декомпиляция finally-блоков в случае их реализации в байткоде через инструкции jsr/ret затруднена тем, что блоки обработки исключений могут быть вложены друг в друга. Семантика обработки исключений предполагает в таком случае последовательное исполнение finally-блоков от различных блоков в порядке их вложенности друг в друга, что может быть неверно в произвольном байткоде. Также в результате оптимизирующих или обфусцирующих преобразований в некоторых точках выхода из try-блока может не выполняться finally-блок. При открытой подстановке finally обратное преобразование затруднено тем, что различные копии finally-блока, подставленные в точки выхода из try и catch блоков, могут претерпеть оптимизирующие или обфусцирующие изменения под воздействием инструмента,

порождающего байткод. При декомпиляции требуется проводить дополнительный анализ и доказывать, что предполагаемые участки кода можно декомпилировать в один finally-блок, что в общем случае алгоритмически неразрешимо.

Рассматривается два случая: открытая подстановка finally и использование инструкций jsr/ret. В первом случае try-блок исходного текста, разделенный в байткоде копиями finally-блока, представлен в таблице исключений несколькими записями (рис. 5). Этот набор записей имеет одинаковые HandlerPC, то есть одинаковые обработчики исключений, и, если не объединить этот набор записей в оригинальную единственную структуру, то невозможно декомпилировать таблицу исключений методом прямой декомпиляции. Однако в разделе 3.2 уже решена проблема декомпиляции таблиц исключений с совпадающими HandlerPC, так что необязательно проводить декомпиляцию finally. Как видно на рис. 6, полученные копии finally-блока можно рассматривать как участки кода, не имеющие никакого отношения к декомпилируемому блоку обработки исключений, и декомпилировать их отдельно.

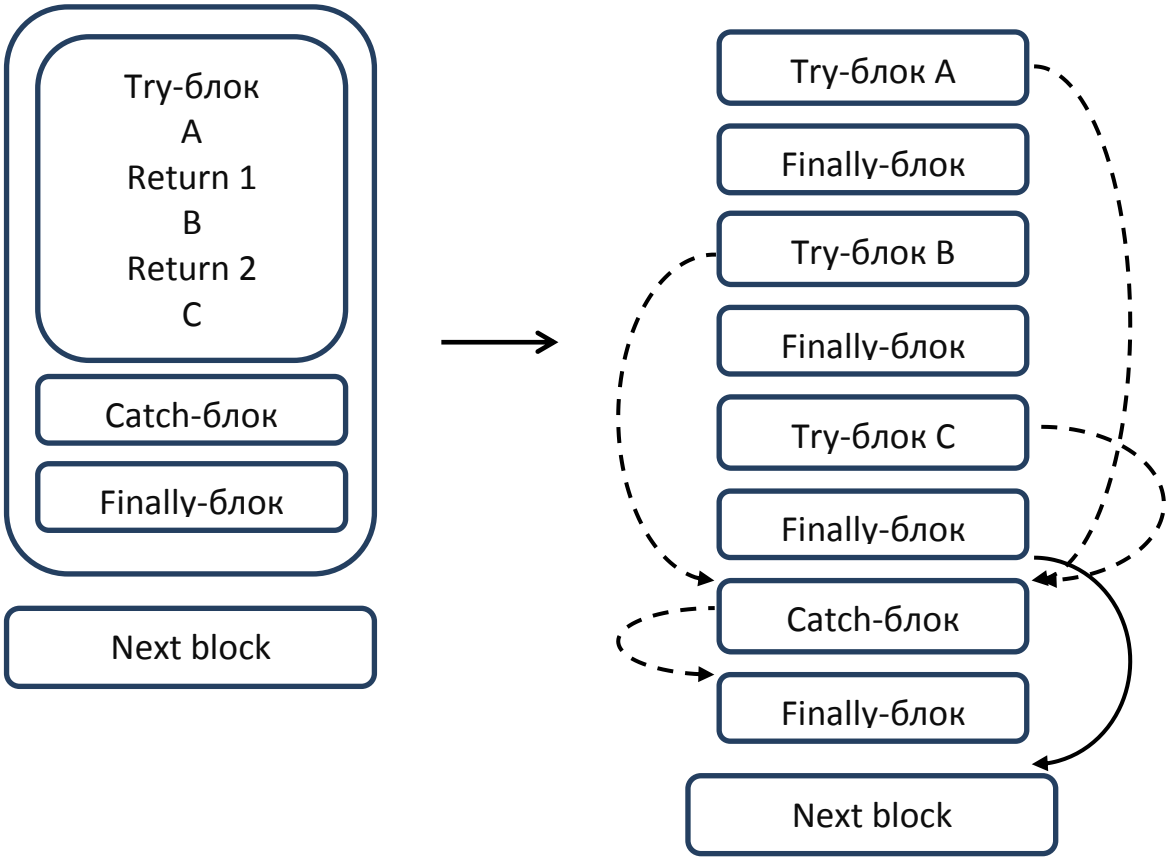


Рис. 5. Результат преобразования блока обработки исключений в результате открытой подстановки finally. Пунктирные дуги соответствуют записям в таблице защищенных интервалов, сплошная дуга — безусловный переход.

Во втором случае, когда finally-блок компилируется с использованием jsr/ret инструкций, для каждой инструкции ret определяется множество соответствующих ей инструкций jsr. Для каждой такой группы, представляющей один finally-блок, заводится локальная переменная local_X. Каждая инструкция jsr_N представляется таким образом:

```
local_X := N;
goto jsr_target;
```

Инструкция ret представляется в AST структурой switch-goto:

```
switch (local_X) {
case 1: goto jsr_1.next;
…
case N: goto jsr_N.next; }
```

Здесь jsr_N.next – оператор, следующий за инструкцией jsr_N. При этом построении нужно так же, как и в алгоритме разрешения проблемы декомпиляции catch-блоков, следить за структурностью создаваемых операторов goto.

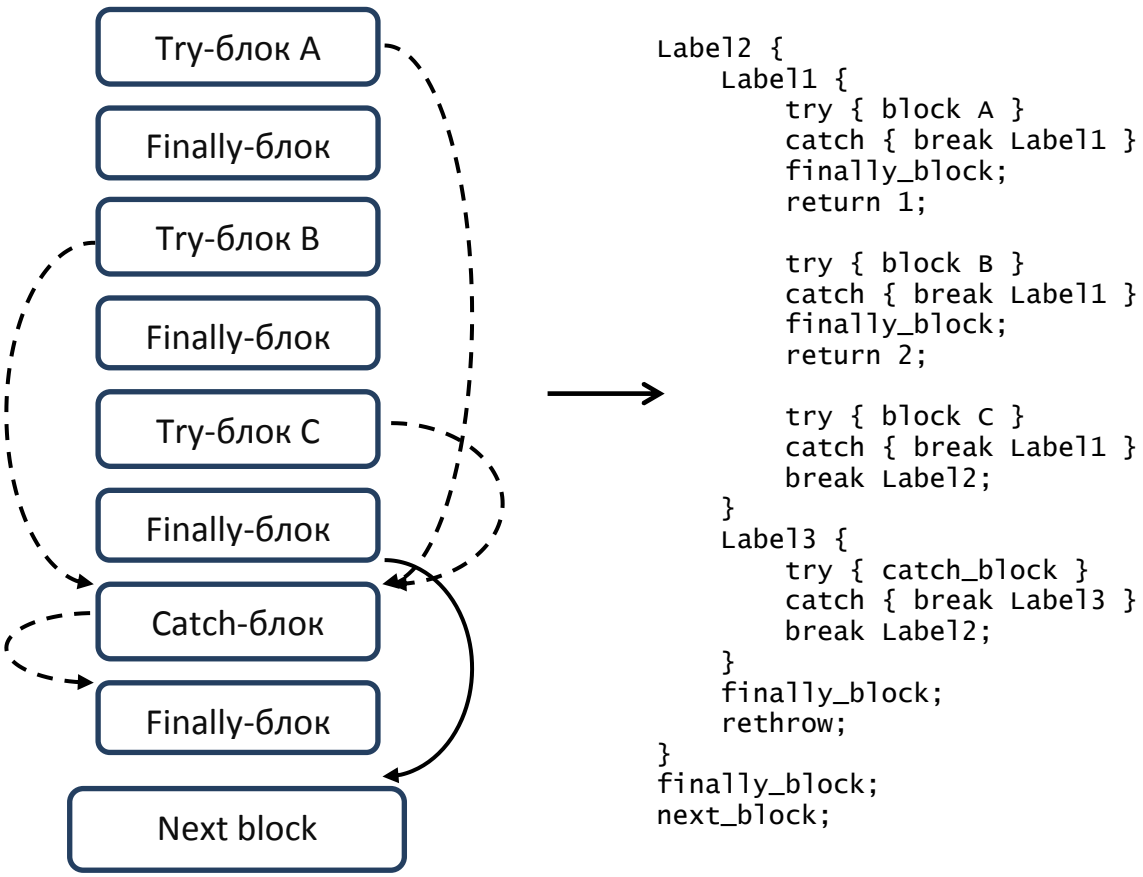


Рис. 6

Алгоритм декомпиляции jsr/ret

1. Выделяется группа операций jsr, соответствующих одной операции ret.
2. Для каждой такой группы заводится дополнительная локальная переменная.

3. Инструкции jsr декомпилируются в оператор, заполняющий значение локальной переменной уникальным значением, и переход на аргумент jsr.
4. Инструкции ret декомпилируются в конструкции switch, каждый case которых по значению локальной переменной осуществляет переход на инструкцию, следующую за инструкцией jsr, соответствующей этому значению.

Утверждение 6: Алгоритм корректен по построению и семантике исполнения jsr/ret инструкций. Трудоемкость алгоритма — O(code), где code — размер декомпилируемого байткода. Это так, потому что первый шаг алгоритма проводится верификатором, который работает до декомпилятора, и можно использовать результаты его работы, а количество всех остальных шагов не может превышать количества инструкций jsr в коде. Емкостная сложность также составляет O(code).

3.4. Общий алгоритм декомпиляции блоков обработки исключений. Общий алгоритм использует все вышеописанный алгоритмы и генерирует структурное представление обработки исключения для произвольной таблицы исключений. Структура AST, соответствующая одному блоку обработки исключений, содержит try-блок и список обработчиков catches. Назовем эту структуру TCB (Try-Catch-Block). Исполнение TCB — это исполнение его блока try. Если во время этого исполнения возникает исключение типа Type, то в списке catches ищется первый обработчик, подходящий по типу исключения. Если он находится, то исключение считается обработанным, и исполняется оператор goto Handler_label. Иначе исключение считается необработанным и передается дальше в структуру, в которую вложена структура TCB.

Общий алгоритм декомпиляции

1. Если таблица нерегулярная, применяем алгоритм приведения таблицы к регулярному виду и, при необходимости, алгоритм уравнивания интервалов.
2. Если есть неструктурные Handler, применяем алгоритм разбиения интервалов.
3. Если в коде есть инструкции jsr/ret, применяем алгоритм декомпиляции jsr/ret.
4. Выделяем в таблице группы интервалов, совпадающих по Start и End. Создаем структуру TCB, try-блок которой получается декомпиляцией интервала [Start, End), а список catches создается из выделенной группы записей в том порядке, в котором они находятся в таблице.
5. Для каждого catch из списка catches порождаем оператор goto, аргумент которого — метка на операторе Handler соответствующего catch.

Трудоемкость и емкостная сложность алгоритма: Используя оценки, полученные ранее, можно сказать, что трудоемкость общего алгоритма декомпиляции равна O(|XT|*|XT| + code). Теоретическая сложность действительно достигает указанной оценки,

но на практике, во-первых, |XT| << code, во-вторых, алгоритм уравнивания интервалов не применяется, в-третьих, алгоритм разбиения интервалов не увеличивает размер таблицы квадратично. К тому же в задаче декомпиляции было бы разумно оценивать трудоемкость в зависимости от размера кода. Таким образом, практической средней оценкой трудоемкости, подтвержденной статистическими данными, будем считать O(code). Емкостная сложность — O(|XT| * |XT| + code), на практике — O(code).

Утверждение о корректности: Семантика обработки исключений декомпилируемой таблицы совпадает с семантикой обработки исключений в структуре TCB, полученной общим алгоритмом декомпиляции. Доказательство приведено в приложении A.

Декларация об универсальности: Общий алгоритм декомпиляции применим для произвольной корректной таблицы защищенных интервалов, т.к. не накладывает на нее никаких ограничений, кроме тех, которые наложены верификатором.

4. Реализация и эксперименты. Общий алгоритм декомпиляции блоков обработки исключений встроен в алгоритм декомпиляции Java-байткода системы Excelsior RVM как дополнительный для уже существующего. Это означает, что он применяется для декомпиляции тех методов, в которых не удается декомпилировать таблицы исключений алгоритмом прямой декомпиляции. Если во время попытки провести прямую декомпиляцию происходит ошибка, то алгоритм возвращается к оригинальным байткоду и таблице исключений и применяет разработанный нами алгоритм. Решение о такой интеграции основано на том, что если есть возможность полного восстановления блоков обработки исключений, то лучше реализовать ее, так как у метода прямой декомпиляции меньше издержки на размер генерируемого внутреннего представления.

Мы сравнили результаты компиляции тестового набора приложений в схеме, в которой работал только алгоритм прямой декомпиляции (эталонный режим), со схемой, когда разработанный нами общий алгоритм декомпиляции был дополнительным для прямого алгоритма (тестируемый режим). Экспериментальные результаты были сведены в таблицу 1, которая содержит следующие колонки: название приложения (приложение), общее количество методов (кол-во методов), количество методов с непустыми таблицами исключений (кол-во таблиц), количество методов, не декомпилированных эталонным режимом (отказы эталона по TCF), количество методов, не декомпилированных обоими режимами по причинам, не связанным с декомпиляцией таблиц исключений (отказы другого рода). В тестируемом режиме для всех приложений были декомпилированы все методы, кроме тех, которые попали в группу «отказов другого рода». В эту группу попадают методы, размер которых полагается компилятором слишком большим для проведения оптимизаций. В тестовый набор были включены крупные Java-приложения и

библиотеки. Среди них можно выделить: Eclipse RCP приложения, систему учета ошибок JIRA, экспериментальную реализацию Java SE — Harmony, реализацию языка Ruby поверх JVM — JRuby, тесты совместимости со спецификацией Java SE — JCK6.

Приложение	**Кол-во методов**	**Кол-во таблиц**	**Отказы эталона по TCF**	**Отказы другого рода**
Eclipse-3.4-jee	381088	25763	1228	7
Eclipse Europa	237196	18669	854	0
Apache Directory Studio	124359	8051	259	0
EAStudio 1.5.1	248045	15784	333	0
Eclipse-3.2	193108	15890	21	0
Escape-K	134765	10128	261	0
MyEclipse_6.0	280787	20442	713	2
MyTourbook 1.5.0	125988	10048	195	0
Relations	105750	8342	8	0
RSSOwl 2.0	61835	4195	144	0
SafiWorkshop 1.0.4	309080	23175	354	1
XMIND 2.3	140961	8274	224	0
Harmony6.0	105223	8562	55	2
Jira 3.12.1	130492	11751	355	0
JRuby-1.1RC1	28256	4381	1998	0
JCK6b	323691	21835	107	0

Таблица 1

На основе этих результатов можно сказать, что невозможность декомпиляции таблицы исключений была основной причиной невозможности декомпиляции байткода в целом. Это подтверждает экспериментально то, что с помощью общего алгоритма декомпиляции таблиц исключений можно построить для произвольного верифицируемого байткода эквивалентное ему структурное представление, т.е. эквивалентную программу на языке Java. Таким образом, на практике подтверждена равномощность Java-байткода и языка Java. Также положительным результатом оказалось то, что для приложения с наибольшим отношением кол-ва методов к количеству отказов эталона — JRuby-1.1RC1, применение тестируемого режима дало прирост производительности порядка 15% за счет лучших оптимизирующих преобразований декомпилированного байткода. Также среди

результатов можно отметить, что система Excelsior RVM в тестируемом режиме прошла Sun JCK 6b (Java Compatibility Kit).

5. Заключение. Началом работ по декомпиляции можно считать работы Дейкстры [6], [7], [13], разработавшего основные положения структурного программирования и развивавшего идею отказа от использования оператора goto. Ему принадлежит формулировка и доказательство теоремы: «Алгоритм любой сложности можно реализовать, используя только три конструкции: следования (линейные), выбора (ветвления) и повторения (циклические).» [6]. Следствием этой теоремы является то, что любой байткод (программу с операторами goto) можно преобразовать в высокоуровневое представление без операторов goto (в нашем случае — только со структурными операторами goto).

Декомпиляцией исполняемого кода процессора i80286 в высокоуровневое представление и затем в исходный текст на языке C занималась Кристина Сайфуентис. Она описала строение и этапы работы абстрактного декомпилятора [2], [3], анализ информационных зависимостей и оптимизацию для указанного процессора [4], а также предложила алгоритмы декомпиляции циклов и условных операторов на основе построения управляющего графа (CFG) [5]. Ее работа была посвящена декомпиляции в исходный текст языка C, поэтому декомпиляция блоков обработки исключений, которых в этом языке нет, не рассматривалась.

Управляющий граф в том виде, в котором он представлен в работах Сайфуентис, не подходит для декомпиляции во внутреннее представление, включающее структуры обработки исключений, т.к. добавление дуг, по которым переходит управление в случае возникновения исключения, нарушает анализ потока управления. В работе Джанг-Ву Йо и Бьенг-Мо Чанг было предложено разбиение управляющего графа, включающего дуги обработки исключений, на Normal Control Flow Graph (NCFG) и Exception-Induced Control Flow Graph (EICFG) [8]. Ими была показана корректность такого разбиения и возможность отдельного анализа NCFG и EICFG.

Авторитетными специалистами в области декомпиляции Java-байткода являются сотрудники канадского университета McGill Джером Мицниковски и Лори Андран. В их статьях рассмотрен алгоритм декомпиляции с использованием внутренних представлений CFG и SET (дерева структурного вложения) [10]. Что касается конкретных проблем с декомпиляцией блоков обработки исключений, то ими также замечено то, что верифицируемые таблицы исключений Java-байткода могут быть неструктурными, что не покрывается предлагаемым ими алгоритмом декомпиляции, однако ими не было предложено какого-либо полного разрешения этих проблем.

Декомпилятор Java-байткода системы Excelsior RVM описан в работе Липского [1]. Он включает в себя алгоритм декомпиляции блоков обработки исключений, основанный на методе прямой декомпиляции с восстановлением полной структуры обработки исключений. Мотивацией для настоящей работы стало то, что с момента реализации исходного алгоритма появилось множество инструментов создания Java-байткода, создающих его таким, что он не удовлетворяет условиям структурности. Из-за этого многие популярные приложения, существующие в виде Java-байткода, стало невозможно декомпилировать в высокоуровневое внутреннее представление.

Данная работа дополняет все предыдущие тем, что в ней классифицированы проблемы декомпиляции блоков обработки исключений, предложены алгоритмы их разрешения, и проведены статистические исследования. Некоторой особенностью подхода является то, что целью было именно построение структурного представления, необходимого для проведения оптимизирующих преобразований, а не исходного текста, как в статьях Сайфуентис и Мицниковски. Такая постановка задачи не требовала генерации структурного представления, позволяющего получить читаемый исходный текст, из которого был получен декомпилируемый байткод. Это позволило беспрепятственно использовать преобразования таблиц исключений. Несмотря на это, из структурного представления, генерируемого описанным алгоритмом декомпиляции, также можно построить и исходный текст на языке Java.

Данная работа вместе с предшествующей работой Липского [1] подтверждает равномощность Java-байткода и языка Java — для произвольного верифицируемого байткода можно построить семантически эквивалентную ему программу на языке Java. Таким образом, задача декомпиляции Java-байткода полностью решена.

Список литературы

1. Липский Н.В. Квалификационная работа магистра «Декомпилятор байт-кода виртуальной Ява машины». НГУ, 1998.
2. Cifuentes, C. A Methodology for Decompilation / C. Cifuentes, K. J. Gough // Proceedings of the XIX Conferencia Latinoamericana deInformatica. – Buenos Aires, Argentina: 1993. – P. 257-266.
3. Cifuentes, C. A structuring algorithm for decompilation / C. Cifuentes // In XIX Conferencia Latinoamericana de Inform'atica. – 1993. – P. 267-276.
4. Cifuentes, C. Interprocedural data flow decompilation / C. Cifuentes // Journal of Programming Languages. – 1996. – Vol. 4. – P. 77-99.
5. Cifuentes, C. Structuring decompiled graphs / C. Cifuentes // In Proceedings of the International Conference on Compiler Construction. – Springer Verlag, 1996. – P. 91-105.

6. Dijkstra, E. W. A Discipline of Programming / E. W. Dijkstra. – 1st edition. – Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.
7. Dijkstra, E. W. Classics in software engineering / E. W. Dijkstra / Ed. by E. N. Yourdon. – Upper Saddle River, NJ, USA: Yourdon Press, 1979. – P. 1-9.
8. Jo, J.-W. Constructing control flow graph that accounts for exception induced control flows for java / J.-W. Jo, B.-M. Chang // Science and Technology, 2003. Proceedings KORUS 2003. The 7th Korea-Russia International Symposium on. – Vol. 2. – 2003. – P. 160-165.
9. Lindholm, T. Java Virtual Machine Specification / T. Lindholm, F. Yellin. – 2nd edition. – Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
10. Miecznikowski, J. Decompiling Java using staged encapsulation / J. Miecznikowski, L. Hendren // Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. – 2001. – P. 368-374.
11. Muchnick, S. S. Advanced compiler design and implementation / S. S. Muchnick. – San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
12. Overview of Excelsior Jet, a high performance alternative to Java virtual machines / V. Mikheev, N. Lipsky, D. Gurchenkov et al. // Proceedings of the 3rd international workshop on Software and performance. – WOSP '02. – New York, NY, USA: ACM, 2002. – P. 104-113.
13. Structured programming / Ed. by O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. – London, UK, UK: Academic Press Ltd., 1972.
14. The Java Language Specification, Third Edition / J. Gosling, B. Joy, G. Steele, G. Bracha. – Prentice Hall, 2005.

Приложение А.

Доказательство утверждения 1

1. Пусть $XT \in XTables$, $ST = Sep_{n,c}(XT)$, $P \in PC/NRF$, $T \in XTypes$. Пусть $\Delta(XT, P, T) = X$. Покажем, что $\Delta(ST, P, T) = X$.

1.1 Если $X \in PC$, то $\exists\, i$: $P \in XT_i.Range$, $T <: XT_i.Type$, причем это i — минимальное, $X = XT_i.Handler$. Если $i < n$, то $\Delta(ST, P, T) = X$, так как $ST_i = XT_i$. Пусть $i = n$. Тогда $P \in XT_n.Range$. Если $P < C$, то $P \in [XT_n.Start, C) = ST_n.Range$. Если $P \geq C$, то $P \in [C, XT_n.End) = ST_{n+1}.Range$. В любом случае, $\Delta(ST, P, T) = ST_n.Handler = XT_n.Handler = X$. Если $i > n$, то $\Delta(ST, P, T) = X$, так как $ST_{i+1} = XT_i$.

1.2 Если $X = \bot$, то $\nexists\, i$: $P \in XT_i.Range$ и $T <: XT_i.Type$. Предположим: $\Delta(ST, P, T) \in PC$. Тогда $\exists\, i$: $P \in ST_i.Range$, $T <: ST_i.Type$. Если $i < n$ или $i > n+1$, то $ST_i = XT_i \Rightarrow \Delta(XT, P, T) = \Delta(ST, P, T) \in PC$ — противоречие. Если $i = n$ или $i = n+1$, то $P \in ST_n.Range \cup ST_{n+1}.Range =$

XT_n.Range. Так как T <: ST_n.Type = ST_{n+1}.Type = XT_n.Type, то Δ(XT, P, T) = Δ(ST, P, T) ∈ PC — противоречие. Δ(ST, P, T) ∉ PC ⇒ Δ(ST, P, T) = ⊥.

Из произвольности P и T следует, что ST ≈ XT.

2. Пусть XT ∈ XTables, UTF(XT_n) = 1, DT = Del_n(XT), P ∈ PC/NRF, T ∈ XTypes. Пусть Δ(XT, P, T) = X. Покажем, что Δ(DT, P, T) = X.

2.1 Если X ∈ PC, то ∃ i: P ∈ XT_i.Range, T <: XT_i.Type, X = XT_i.Handler. (UTF(XT_n) = 1) ⇒ (∀ C ∈ XT_n.Range: RF(C) = 0) ⇒ (XT_n.Range ⊂ NRF) ⇒ (P ∉ XT_n.Range) ⇒ (i ≠ n). Если i < n, то DT_i = XT_i. Если i > n, то DT_{i-1} = XT_i. В любом случае, Δ(DT, P, T) = X.

2.2 Если X = ⊥. Предположим: Δ(DT, P, T) ∈ PC. Тогда ∃ k: P ∈ DT_k.Range. ∀ k = 0,..., |DT|: ∃ i такое, что DT_k = XT_i. (T <: DT_k.Type) ⇒ (∃ i: P ∈ XT_i.Range, T <: XT_i.Type) ⇒ (X = Δ(XT, P, T) = XT_i.Handler ≠ ⊥). Противоречие.

Из произвольности P и T следует, что DT ≈ XT.

3. Пусть $Del_{n,X}^{-1}$ — преобразование, обратное Del_n, удаляющему интервал X под номером n, такой, что UTF(X) = 1. Это преобразование эквивалентное в силу доказанного выше. Пусть $Sep_{n,C}^{-1}$ — преобразование, обратное $Sep_{n,C}$. Оно является эквивалентным, если C = XT_n.End = XT_{n+1}.Start. Пусть Int = <XT_n.End, XT_{n+1}.Start, XT_n.Type, XT_n.Handler>. Тогда UTF(Int) = 1, т.к. [XT_n.End, XT_{n+1}.Start) ⊂ NRF. Тогда эквивалентным является следующее преобразование:

$Concat_n = Sep_{n,(XT\_n+1).StartPC}^{-1}(Sep_{n,XT\_n.EndPC}^{-1}(Del_{n+1,Int}^{-1}))$. ■

Доказательство утверждения 2

Подтвердим оценку трудоемкости алгоритма. Трудоемкость первого этапа (сортировка) равна O(|XT|*log(|XT|)). Трудоемкость второго этапа равна O(|XT|), т.к. для каждого интервала верно, что он может попасть на стек только один раз и только один раз может быть оттуда снят, при этом на каждом шаге алгоритма осуществляется снятие одного интервала со стека. Таким образом, временная сложность алгоритма составляет O(|XT|*log(|XT|)). Емкостная сложность алгоритма составляет O(|XT|) т.к. это максимальный размер, которого может достигать стек. Докажем теперь, что алгоритм действительно проверяет, является ли таблица регулярной в смысле определения, данного в разделе 5.1.

Пусть есть интервалы A = XT_i и B = XT[j], пересекающиеся друг с другом без вложения или неправильно вложенные, причем i минимальное, а j минимальное для i. В любом случае A и B имеют непустое пересечение. Без ограничения общности предполагаем, что A.Start <= B.Start. Тогда в результате сортировки A находится в отсортированном массиве раньше B. Таким образом, A будет рассматриваться на втором шаге алгоритма раньше, чем B и, к моменту рассмотрения B будет еще находиться на стеке.

Действительно, если это не так, то между рассмотрениями A и B был рассмотрен некоторый интервал XT_k (i < k < j), такой что A не пересекался с XT_k, т.к. только в этом случае он мог быть снят со стека. В силу сортировки массива XT это означает, что XT_k.Start >= A.End. Также в силу сортировки массива XT, XT_k.Start <= B.Start. A.End <= XT_k.Start <= B.Start — противоречие с непустой пересеченностью A и B. Таким образом, действительно A находится на стеке в момент рассмотрения B.

Если A находится на стеке, то все элементы, находящиеся выше A, вложены в него, причем правильно вложены, т.к. их добавление не вытолкнуло A со стека, что означает, что они пересекались с A, а минимальность нерегулярности (A – B) гарантирует, что они не остановили алгоритм.

Когда B перестанет очищать стек, то на верхушке стека будет находиться некоторый XT_k (i <= k < j). При этом XT_k и B имеют непустое пересечение. Если k = i, то нерегулярность (A – B) определяется сразу же.

Пусть i < k. Если B неправильно вложен в A (т.е. в оригинальной таблице B имел больший номер, чем A), и при этом он имеет непустое пересечение с XT_k, то это означает и то, что он неправильно вложен в XT_k (или пересекается без вложения с ним), так как в силу правильного вложения XT_k в A, номер в оригинальной таблице XT_k меньше номера в оригинальной таблице A. Если же B пересекается без вложения с A, то, в силу вложения XT_k в A и пересечения B и XT_k, B пересекается без вложения с XT_k.

Таким образом, B пересекается без вложения или неправильно вложен в XT_k, находящийся на верхушке стека, что и будет определено алгоритмом. ■

Доказательство утверждения о корректности общего алгоритма декомпиляции.

Для шагов 1–3 общего алгоритма уже доказано, что таблица исключений, преобразованная этими шагами, эквивалентна оригинальной таблице исключений и что таблица исключений, полученная после 3-ого шага – регулярная.

Таким образом, надо доказать для шагов 4–5, что семантика обработки исключений для таблицы, поданной на вход 4-ому шагу, совпадает с семантикой обработки исключений в структуре TCB, получаемой на выходе.

Рассмотрим все возможные пути обработки исключений.

1. Отсутствие исключений.

В таком случае в структуре TCB выполнится try-блок. Затем, если finally-блок был скомпилирован jsr/ret инструкциями, то произойдет переход на операторы, в которые была декомпилирована инструкция jsr. Если finally-блок был скомпилирован открытой подстановкой, то он исполнится как декомпилированный код, следующий за структурой

ТСВ. Если же finally-блока не было, то исполнение также перейдет на операторы, следующие за ТСВ, что эквивалентно исполнению на оригинальном байткоде.

2. Обработка исключения.

Исполнение try-блока прервется на моменте возникновения исключения. Условие обработки исключения структурой ТСВ состоит из условия вложения инструкции, вызвавшей исключение, в try-блок этой структуры и условия наличия среди обработчиков catches обработчика, тип которого совместим по присваиванию с типом возникшего исключения. Все структуры, вложенные в рассматриваемую нами, не обработали исключение. Это означает, что рассматриваемая нами структура — первая подходящая по условию обработки исключения. Таким образом, из условия правильной вложенности всех интервалов рассматриваемая структура ТСВ соответствует первой записи таблицы исключений, подходящей под те же самые условия. В результате обработки исключения исполнение перейдет на инструкцию с адресом Handler, что эквивалентно обработке исключений в оригинальном байткоде. Finally-блок также будет исполнен, независимо от того, каким инструментом он был представлен в байткоде: jsr/ret или открытая подстановка.

3. Пропуск исключения.

Если в оригинальном байткоде предполагалось, что у блока обработки исключений есть finally-блок , то среди обработчиков обязательно появляется обработчик с нулевым значением CatchType, что означает произвольный тип исключения. Этот обработчик содержит finally-блок (реализованный jsr/ret или открытой подстановкой) и инструкцию throw, перевыбрасывающую исключение и тем самым продолжающую его обработку. Таким образом, пропуск исключения структурой — это частный случай обработки исключения. При этом пропущенное исключение будет обработано (или также пропущено) объемлющей структурой. Рассматриваемая же структура также из свойства правильной вложенности соответствует в таблице исключений записи с меньшим номером, чем объемлющая, что эквивалентно обработке исключений в оригинальном байткоде. ■

UDK: 004.4'422

Title: Exception handling blocks building at Java-bytecode decompilation

Author(s):

Solovyov Vladimir Valeryevich (A.P. Ershov Institute of Informatics Systems, Novosibirsk State University),

Lipsky Nikita Valeryevich (A.P. Ershov Institute of Informatics Systems)

Annotation: Java bytecode decompilation is a process of reverse translation that restores Java source code by the corresponding bytecode. Java bytecode is an intermediate representation based on abstract stack machine. It may have arbitrary control flow graph, whereas the Java language contains control structures that always form a strict hierarchy. Decompilation aims at restoring all control structures, including Java exception handling blocks. In the Excelsior RVM (Java Virtual Machine with a static compiler), bytecode is decompiled in a structural intermediate representation for further optimization. When building exception handling blocks, the Excelsior RVM's compiler assumes that the bytecode must be emitted by the standard Java source to bytecode compiler and uses a few heuristics to make reverse transformation. However, it is not always possible if the bytecode is produced by other instruments. This paper presents a decompilation algorithm that produces exception handling blocks given any correct bytecode. The algorithm has been implemented, integrated into the Excelsior RVM and tested on real-world applications.

Keywords: compilers, decompilers, Java, Java bytecode, exceptions handling, exceptions handling blocks, exception tables