администрации НИИ ИТ СПбГУ за финансовую поддержку издания этого выпуска.

Наконец, мы рады сообщить, что с этого года тексты статей сборников серии «Системное программирование» доступны в сети Интернет по адресу http://www.sysprog.info. Там же можно ознакомиться с требованиями, предъявляемыми к статьям, представленным для публикации, и новостями, связанными с подготовкой и распространением нашего издания.

А. Н. Терехов, Д. Ю. Булычев

Инкрементальный статический анализ кода на основе разложения отношений

А. В. Кириллин
sashakir@inbox.ru

В статье рассматривается подход к инкрементализации статического анализа кода, позволяющий выполнять эффективный пересчет результатов анализа при малых изменениях кода. Для формализации методов анализа был разработан язык `ERML`, оперирующий отношениями. В качестве внутреннего представления данных выбрана структура BDD (Binary Decision Diagrams). На языке `ERML` реализован набор алгоритмов статического анализа для применения в рамках инструмента для понимания программ.

Введение

Задача статического анализа кода заключается в выяснении различных свойств программы без ее исполнения. До недавнего времени статический анализ применялся в основном в оптимизирующих компиляторах. Сейчас все более востребованными становятся инструменты, облегчающие процесс разработки программ. Большинство этих инструментов так или иначе основаны на анализе разрабатываемого кода. Вот лишь некоторые примеры таких инструментов: профиляторы, средства для понимания программ[1], средства построения тестового покрытия, средства верификации, инструменты рефакторинга, составители метрик. Причем если раньше весь набор таких инструментов обычно поставлялся как набор

[1] В англоязычной литературе средства для понимания программ известны как *program understanding tools*.

отдельных программ, то сейчас существует тенденция к интегрированию всех необходимых средств в единую среду разработки программ.

Разрабатывая средство статического анализа кода для интегрированных сред программирования, необходимо учитывать то, что анализируемый код находится в стадии разработки и подвержен частым изменениям. Постоянно производить анализ «с нуля» при каждом изменении кода не совсем оправдано хотя бы потому, что большинство алгоритмов анализа работают очень долго и требуют много ресурсов. Отсюда возникает проблема эффективного пересчета результатов анализа по изменившемуся коду — проблема *инкрементализации* анализа.

Задача данной работы — разработать общий метод инкрементального анализа, не зависящий ни от применяемых методов анализа, ни от видов дальнейшего использования его результатов. Для этого необходимо создать язык, на котором можно было бы запрограммировать набор основных алгоритмов статического анализа кода. Для того чтобы данная работа имела практическую пользу, нужно разработать инструмент, интегрирующий возможности статического анализа кода в одну из широко используемых сред программирования. Этот инструмент должен быть универсальным и обеспечивающим возможность отображения результатов любого запрограммированного алгоритма анализа. Кроме того, необходимо выявить преимущества инкрементального подхода к статическому анализу, то есть сравнить время работы и размер потребляемых ресурсов при инкрементальном пересчете анализа и пересчете анализа «с нуля».

1. Существующие подходы

В данном разделе представлен обзор существующих подходов к решению задач, связанных с инкрементальным статическим анализом кода.

1.1. Анализ кода

Существует огромное количество литературы, посвященной анализу кода, например известные учебники [1, 10]. Основные задачи анализа, применяемые как в инструментах для понимания программ, так и в большинстве других приложений статического анализа, — это *анализ указателей*[2] и *анализ срезов*[3]. Анализ указателей подробно изучен в работах [4, 19, 22, 9]. Среди прочих работ по анализу указателей можно выделить две [12, 2], на которые так или иначе опираются все остальные исследования в этой области. Обзор основных методов анализа срезов представлен в [13]. В этой работе выделены два вида анализа срезов: *статический анализ срезов* и *динамический анализ срезов*. Интересный алгоритм динамического анализа срезов предложен в [21].

В статье [3] рассматриваются способы оптимизации производительности инструментов для понимания программ. Авторы предлагают перейти от анализа всей программы целиком к анализу только той части кода, которая необходима пользователю в данный момент[4]. Но как показала практика, на реальных программах и реальных методах анализа такой подход не работает, потому что разные части программы обычно бывают очень тесно связаны между собой, и, например, чтобы найти места, на которые может ссылаться указатель, необходимо проанализировать 75–90% всего кода программы.

1.2. Анализ с помощью отношений

Было сделано немало попыток использовать отношения для записи алгоритмов анализа. Например, [6] предлагает язык *RML*, который является расширением языка логики первого порядка. В [8] описан язык *Rscript*, оперирующий только бинарными отношениями, использующийся для написания простейших анализаторов кода. В нем введены некоторые специфические операторы для работы с отношениями. В работе [19] автор использует язык *Datalog* [14] для анализа указателей. Существует целый ряд методов анализа объектно-ориентированного кода, основанных на отношениях: анализ классов [5], поиск объектно-ориентированных шаблонов [11] и др. В [16] реализован браузер исходного кода, поддерживающий сложные реляционные запросы на основе *TyRuBa* [17] — логического языка, напоминающего Пролог. Также есть некоторые попытки использования SQL [7] для анализа кода.

[2] В англоязычной литературе анализ указателей известен как *points-to analysis* и *pointer analysis*.

[3] В англоязычной литературе анализ срезов известен как *slices analysis*.

[4] В англоязычной литературе такой подход известен как *demand-driven computation*.

1.3. Инкрементальные алгоритмы анализа

Область инкрементального анализа кода до сих пор довольно слабо изучена, однако и тут есть некоторые результаты. В работе [20] описывается инкрементальная версия *анализа псевдонимов*[5], близкого к анализу указателей. Хотя здесь и не описан общий способ инкрементализации любого алгоритма анализа, подход, предложенный в статье, допускает обобщение и на другие методы анализа, построенные по итеративной схеме. При изменении частей кода автор предлагает ослабить полученные оценки для этих частей кода до достижения безопасного решения. Затем процесс итераций перезапускается, пока алгоритм не сойдется. При ослаблении оценок для измененных частей кода важно ослабить их как можно меньше, получив тем не менее при этом безопасное решение, так как более слабые оценки приводят к большему количеству итераций алгоритма, тем самым уменьшая его эффективность.

В работе [15] предлагается иной подход к анализу указателей. Согласно нему каждый метод (процедура) в коде анализируется отдельно и для них строятся функции перехода, описывающие, как изменился граф указателей после вызова метода. Если анализируемый метод содержит вызовы других методов, их функции перехода «подставляются» в функцию перехода вызывающего метода. Понятно, что такой подход к анализу указателей инкрементализовать значительно проще, так как локальные изменения кода влияют лишь на небольшое количество функций перехода. Подробный алгоритм инкрементального анализа указателей также описан в этой статье.

Если инкрементальный анализ кода еще не нашел широкого применения, то инкрементальный синтаксический разбор кода (parsing) уже давно стал неотъемлемой частью любой среды программирования. В статье [18] описан способ построения инкрементальных LR-парсеров, основанных на технологии GLR (Generalized LR Parsing).

2. Анализ на основе инкрементальных разложений

Для записи алгоритмов анализа кода был создан новый язык, расширяющий возможности языка `RML` [6]. Далее будем называть

[5] В англоязычной литературе анализ псевдонимов известен как *alias analysis*.

этот язык *ERML (Extended RML)*. Это скрипто-подобный язык, основные конструкции в нем — операторы присваивания и циклы. Каждый алгоритм анализа будет представляться в виде скрипта (последовательности операций над отношениями). Какие-то отношения являются базовыми и даются скрипту на вход. Результатом работы скрипта также будут отношения.

2.1. Отношения и операции над ними

Скрипты на языке `ERML` выполняют преобразования отношений. Приведем несколько примеров наиболее часто используемых отношений.

- $ReturnStatement(S)$ — множество терминальных операторов. Элемент s принадлежит $ReturnStatement$, если оператор s представляет собой оператор возврата `return`.
- $SuperType(T, T)$ — иерархия типов. Элемент (t_1, t_2) принадлежит $SuperType$, если тип t_1 — наследник t_2.
- $LocalVarDeclaration(V, M, T)$ — объявления локальных переменных. Элемент (v, m, t) принадлежит $LocalVarDeclaration$, если локальная переменная v типа t объявлена в методе m.

Работа с отношениями напоминает работу с предикатами в логике первого порядка: отношения соответствуют предикатам, операции объединения, пересечения, проекции над отношениями соответствуют операциям логического «или», «и», квантору существования соответственно. Поэтому за основу синтаксиса языка `ERML` был взят язык предикатов. Далее подробно опишем основные операции над отношениями языка `ERML`:

- операция *объединения*: `A( x ) | B( x );`
- операция *пересечения*: `A( x ) & B( x );`
- операция *разности*: `A( x ) - B( x );`
- операция *импликации*: `A( x ) => B( x );`
- операция *дополнения*: `! A( x );`
- операция EX[6];

[6] В реляционной терминологии операция EX известна как оператор проекции.

- операция *FA*.

Пример операции EX:

EX`[ s3: S ].( StrictDom( s1, s3 ) & Dom( s3, s2 )).`

Смысл этого оператора таков: множество, определенное в пространстве, «натянутом» на атрибуты `s1`, `s2`, `s3`, проецируется на подпространство, «натянутое» на атрибуты `s1` и `s2`.

Операция EX соответствует квантору «$\exists$» в логике первого порядка.

Пример операции *FA*:

FA`[ s3: S ].CFG( s3, s2 ) => Dom( s1, s3 ).`

Синтаксически эта операция аналогична предыдущей, а ее семантика такова: «FA`[...].R(...)`» эквивалентно «`!`EX`[...].(! R(...))`».

Операция FA соответствует квантору «$\forall$» в логике первого порядка.

2.2. Инкрементальные разложения отношений и операций

Здесь предлагается иной способ представления отношений, который позволяет реализовывать операции над отношениями инкрементально. Будем представлять отношения, которыми оперирует язык `ERML`, кортежем

$$(R_{base}, R_{add}, R_{sub}, R_{result}). \tag{1}$$

Элементы $R_{base}, R_{add}, R_{sub}, R_{result}$ — это обычные отношения, определенные выше в разделе 2.1. На самом деле одно из них — R_{result} — лишнее, так как всегда выполняется инвариант

$$R_{result} = (R_{base} \cup R_{add}) \setminus R_{sub}, \tag{2}$$

причем значение отношения R_{result} как раз и определяет значение представляемого всем кортежем (1) инкрементального отношения. Остальные три отношения представляют собой *инкрементальное разложение* отношения R_{result}. Благодаря этому разложению

и удается ускорить производительность операций над отношениями при икрементальном пересчете анализа. Грубо говоря, R_{base} — это R_{result} в «старом» анализе, R_{add} — это те элементы отношения, которые добавились к старому отношению, а R_{sub} — те элементы, которые присутствовали в старом отношении, но отсутствуют в новом. Поскольку, вообще говоря, таких разложений может быть очень много, вводятся дополнительные ограничения:

$$R_{add} \cap R_{sub} = \varnothing, \tag{3}$$
$$R_{sub} \subseteq R_{base}, \tag{4}$$
$$R_{add} \cap R_{base} = \varnothing. \tag{5}$$

Примечание: условие (3) следует из (4) и (5), но здесь оно оставлено для большей ясности и симметричности.

Таким образом, если зафиксировать два параметра — R_{base} и R_{result} — такое разложение будет единственно.

Лемма 1. *Пусть R_{result} и R_{base} — отношения. Тогда существуют и единственны отношения R_{add} и R_{sub}, удовлетворяющие разложению (2) и ограничениям (3), (4) и (5).*

Более того, при этих ограничениях порядок операций объединения и вычитания в формуле (2) становится не важным, и мы можем опустить скобки:

$$R_{result} = R_{base} \cup R_{add} \setminus R_{sub}. \tag{6}$$

Определение 1. *Инкрементальным разложением отношения R по отношению R_{base} будем называть разложение (2), при котором выполнены ограничения (3), (4) и (5).*

Определение 2. *Инкрементальным разложением операции $\oplus$ над отношениями R_1 и R_2 будем называть инкрементальное разложение отношения $R^1_{result} \oplus R^2_{result}$, представленное в виде*

$$\begin{aligned} R^1_{result} \oplus R^2_{result} = \\ & (R^1_{base} \oplus R^2_{base}) \cup \\ & F(R^1_{base}, R^1_{add}, R^1_{sub}, R^2_{base}, R^2_{add}, R^2_{sub}) \setminus \\ & G(R^1_{base}, R^1_{add}, R^1_{sub}, R^2_{base}, R^2_{add}, R^2_{sub}), \end{aligned} \tag{7}$$

где $R^1_{result} = R^1_{base} \cup R^1_{add} \setminus R^1_{sub}$ *и* $R^2_{result} = R^2_{base} \cup R^2_{add} \setminus R^2_{sub}$ *— инкрементальные разложения отношений* R_1 *и* R_2 *соответственно, причем ни в выражении* F*, ни в выражении* G *не должны встречаться операции вида* $R^1_{base} \oplus R^2_{base}$*,* $R^1_{base} \cup R^2_{base}$*,* $R^1_{base} \cap R^2_{base}$ *и т.д. Естественно, подразумевается выполнение условий (3)–(5) для выражений* F *и* G*.*

Теперь мы можем определить основные операции над отношениями в терминах инкрементального разложения.

Инкрементализация дополнения:

$$\overline{r_{base} \cup r_{add} \setminus r_{sub}} = \overline{r_{base}} \setminus r_{add} \cup r_{sub}. \tag{8}$$

То есть если отношение R_1 было инкрементально разложено в виде $R_1 = r^1_{base} \cup r^1_{add} \setminus r^1_{sub}$, то дополнение этого отношения $R_2 = \overline{R_1}$ может быть инкрементально разложено в виде $R_2 = r^2_{base} \cup r^2_{add} \setminus r^2_{sub}$, где $r^2_{base} = \overline{r^1_{base}}$, $r^2_{add} = r^1_{sub}$, $r^2_{sub} = r^1_{add}$. Очевидно, что r^2_{base}, r^2_{add} и r^2_{sub} будут удовлетворять ограничениям (3)–(5).

Инкрементализация объединения:

$$\begin{aligned}(l_{base} \cup l_{add} \setminus l_{sub}) \cup (r_{base} \cup r_{add} \setminus r_{sub}) = \\ (l_{base} \cup r_{base}) \cup (l_{add} \cup r_{add}) \setminus \\ (l_{sub} \cup r_{sub} \setminus l_{result} \setminus r_{result}).\end{aligned} \tag{9}$$

Таким образом, если отношения R и L были инкрементально разложены в виде $R = r_{base} \cup r_{add} \setminus r_{sub}$ и $L = l_{base} \cup l_{add} \setminus l_{sub}$, то объединение может быть инкрементально разложена в виде $U = u_{base} \cup u_{add} \setminus u_{sub}$, где $u_{base} = l_{base} \cup r_{base}$, $u_{add} = l_{add} \cup r_{add}$, $u_{sub} = l_{sub} \cup r_{sub} \setminus l_{result} \setminus r_{result}$. Проверка условий (3)–(5) довольно трудоемка и здесь не приводится.

Инкрементализация оператора EX:

$$\begin{aligned}EX[].(r_{base} \cup r_{add} \setminus r_{sub}) = \\ EX[].r_{base} \cup EX[].r_{add} \setminus \\ (EX[].r_{sub} \setminus EX[].(r_{result} \cap EX[].r_{sub})).\end{aligned} \tag{10}$$

Проецируемый атрибут и его тип не так важны, поэтому они не указаны в операторах EX.

Инкрементальные разложения остальных операций над отношениями довольно громоздки и здесь не приведены. Некоторые из этих операций, например операции пересечения, разности, импликации и FA, выражаются через операции объединения, дополнения и EX.

Итак, мы можем сформулировать следующую теорему:

Теорема 1. *Все операции над отношениями имеют инкрементальные разложения.*

2.3. Инкрементальный анализ

Ранее упоминалось, что инкрементальное разложение позволяет значительно ускорить операции над отношениями. Рассмотрим это более подробно. Пусть имеется *последовательность анализируемых данных*

$$D_1, D_2, \ldots, D_n.$$

Это могут быть исходные коды программ, базы данных и другие источники информации, которую нужно проанализировать. Будем считать, что элементы этой последовательности близки друг к другу. Формально это описать невозможно, но можно пояснить на примере. Типичный пример — последовательность изменений в программе, т. е. одна и та же программа на разных этапах ее разработки. Поскольку обычно программа разрабатывается последовательно (а не переписывается каждый раз заново), то можно полагать, что соседние версии (и не только соседние) будут близки друг к другу.

Нам необходимо проанализировать подряд каждый элемент этой последовательности, то есть, говоря формально, необходимо построить *последовательность анализов* (множеств отношений)

$$\{R_{11}, R_{12}, \ldots, R_{1m}\}, \{R_{21}, R_{22}, \ldots, R_{2m}\}, \ldots, \{R_{n1}, R_{n2}, \ldots, R_{nm}\},$$

соответствующих исходной последовательности данных. Основная идея здесь — это представление каждого отношения R_{ij} в инкрементальном виде по отношению R_{i-1j}. То есть каждое отношение R_{ij} будет разложено в виде $R_{ij} = R_{i-1j} \cup R^{add}_{ij} \setminus R^{sub}_{ij}$. Здесь R_{i-1j} — это то же самое отношение для анализа предыдущего элемента данных (D_{i-1}), R^{add}_{ij} — это то, что «добавилось» к отношению R_j, а R^{sub}_{ij} —

то, что «исчезло» из отношения. Выигрыш здесь состоит в том, что размеры отношений R_{ij}^{add} и R_{ij}^{sub} на порядок меньше, чем размер R_{i-1j}, а вычислять требуется только R_{ij}^{add} и R_{ij}^{sub}, в то время как R_{i-1j} уже было посчитано на предыдущем этапе.

Рассмотрим зависимости между отношениями в контексте одного анализа.

Определение 3. *Будем говорить, что отношение R_1 зависит от отношения R_2, если для того, чтобы вычислить отношение R_1, необходимо иметь отношение R_2.*

Определение 4. *Будем говорить, что отношение R — базовое, если для его вычисления необходим только соответствующий элемент данных D_i.*

Для простоты будем исключать возможность циклической зависимости между отношениями. Таким образом граф зависимостей между отношениями внутри анализа является дэгом. То есть существует порядок p, такой, что в последовательности отношений $\{R_{p(1)}, R_{p(2)}, \ldots, R_{p(m)}\}$ каждое отношение зависит только от предыдущих отношений.

Пусть для элемента данных D_{i-1} анализ $\{R_{i-11}, R_{i-12}, \ldots, R_{i-1m}\}$ проведен полностью. Теперь необходимо построить анализ для следующего элемент данных D_i, используя D_{i-1} и анализ $\{R_{i-11}, R_{i-12}, \ldots, R_{i-1m}\}$. Здесь задача разбивается на две — вычисление базовых отношений и вычисление остальных отношений.

2.3.1. Вычисление базовых отношений

Если разбить элемент данных на несколько долей, то при небольшом изменении данных (а мы договаривались рассматривать только такие изменения) изменению подвержена только небольшая часть долей. Чтобы вычислить базовое отношение, необязательно рассматривать все доли, достаточно рассмотреть только изменившиеся, а информацию об остальных долях можно взять из предыдущего анализа. Более формально, предположим, элемент данных D_i представляет собой разбиение $\{D_{ij}\}_{j=1}^{k}$, и базовые отношения аддитивны относительно этого разбиения[7], т.е. $R(D_i) = \bigcup_{j=1}^{k} R(D_{ij})$.

[7]Здесь R рассматриается не как отношение, а как функция, действующая из области элементов данных D в отношения.

Пусть нам необходимо вычислить базовое отношение R_{ik}. Пусть множество C — это множество изменившихся долей. Тогда

$$R_{ik}(D_i) = \bigcup_{j \notin C} R_{ik}(D_{ij}) \cup \bigcup_{j \in C} R_{ik}(D_{ij}) = (R_{i-1k}(D_{i-1}) \setminus \bigcup_{j \in C} R_{i-1k}(D_{i-1j})) \cup \bigcup_{j \in C} R_{ik}(D_{ij}). \quad (11)$$

То есть, говоря простыми словами, чтобы вычислить базовое отношение по изменившемуся элементу данных, нужно из отношения по старому элементу вычесть отношения по изменившемся долям в старом элементе и добавить отношения по изменившемся долям в новом элементе. Ускорение здесь достигается за счет того, что количество изменившехся долей намного меньше общего количества долей, поэтому вычислять последние два объединения в формуле (11) намного быстрее, чем объединение по всем долям.

Если перейти от общей задачи анализа данных к задаче анализа исходного кода, то здесь можно добиться еще большего ускорения, если принимать в расчет то, что код в большинстве случаев меняется локально, в границах одной доли, хотя это зависит от того, как выбирать эти доли. Например, в объектно-ориентированных языках разумнее всего выбирать в роли долей классы, в модульных языках — модули и т. д. Также можно в роли долей выбирать процедуры. Важно правильно выбрать «гранулярность» долей. При увеличении размера долей уменьшается количество элементов в объединении, однако при изменении в доле придется перестраивать базовые отношения по всей доле. С другой стороны, если уменьшать доли, размер перестраиваемых отношений будет меньше, но будет больше затрат на разбиение всей программы на такие доли.

Формула (11) напоминает инкрементальное разложение отношения R_{ik}. Чтобы оно стало таковым (а это понадобится в дальнейшем), необходимо обеспечить выполнения условий (3)–(5). Введем обозначения:

- $\overline{R_{i-1k}} = \bigcup_{j \in C} R_{i-1k}(D_{i-1j})$ — отношения по изменившемся долям в старом элементе;
- $\overline{R_{ik}} = \bigcup_{j \in C} R_{ik}(D_{ij})$ — отношения по изменившемся долям в новом элементе.

Тогда получаем инкрементальное разложение отношения R_{ik}:

$$R_{ik} = R_{ik}^{base} \cup R_{ik}^{add} \setminus R_{ik}^{sub}, \tag{12}$$

где

$$\begin{array}{rcl} R_{ik}^{base} & = & R_{i-1k}, \\ R_{ik}^{add} & = & \overline{R_{ik}} \setminus \overline{R_{i-1k}}, \\ R_{ik}^{sub} & = & \overline{R_{i-1k}} \setminus \overline{R_{ik}}. \end{array} \tag{13}$$

Разложение (12) удовлетворяет условиям (3)–(5) по построению.

2.3.2. Вычисление остальных отношений

Перейдем теперь к инкрементальному вычислению отношений, не являющихся базовыми. Как было установлено выше, существует такой порядок отношений $\{p_i\}$, что в последовательности отношений $\{R_{p(1)}, R_{p(2)}, \ldots, R_{p(m)}\}$ каждое отношение функционально зависит от предыдущих. То есть для каждого $R_{p(i)}$ существует некая функция $R_{p(i)} = F(R_{p(1)}, R_{p(2)}, \ldots, R_{p(i-1)})$, в составе которой могут быть отношения и операции над ними. Пример такой функции:

```
Dom( s1, s2 ) := BranchStatement( s1 ) & ( s1 = s2 ) |
  BranchStatement( s2 ) & ! Start( s2 ) &
  FA[ s3: S ].CFG( s3, s2 ) => Dom( s1, s3 );
```

Дальнейшая цель — представить эту функцию в виде так называемого трехадресного кода — последовательности операторов, каждый из которых в общем случае имеет вид $\mathcal{R}_i = \mathcal{R}_j \oplus \mathcal{R}_k$, т. е. разбить сложную функцию на атомарные операторы, введя, возможно, некоторые промежуточные отношения. Например, приведенный выше пример вычисления отношения Dom разбивается в такую последовательность операторов:

```
R1( s1, s2 ) := ( s1 = s2 );
R2( s2 ) := ! Start( s2 );
R3( s1, s2, s3 ) := CFG( s3, s2 ) => Dom( s1, s3 );
R4( s1, s2 ) := FA[ s3: S ].R3( s1, s2, s3 );
R5( s1, s2 ) := BranchStatement( s1 ) & R1( s1, s2 );
R6( s2 ) := BranchStatement( s2 ) & R2( s2 );
R7( s1, s2 ) := R6( s2 ) & R4( s1, s2 );
Dom( s1, s2 ) := R5( s1, s2 ) | R7( s1, s2 );
```


Если предположить, что отношения, стоящие слева, не переиспользуются, и промежуточные отношения рассматривать как обычные небазовые отношения, последовательность трехадресных инструкций для вычисления анализа элемента D_i можно представить в виде $\{\mathfrak{R}_{i1}, \mathfrak{R}_{i2}, \ldots, \mathfrak{R}_{il}\}$, где $\mathfrak{R}_{ij}$ представляет отношение, являющееся базовым, или отношение, стоящее слева в какой-то инструкции.

Определение 5. *Пусть j_b — это количество базовых отношений, а $\mathcal{R}_{ij}$ — это отношение, стоящее слева в j-й трехадресной инструкции, посчитанное для i-го элемента данных. Тогда будем называть расширенной последовательностью анализов последовательность*

$$\{\mathfrak{R}_{11}, \mathfrak{R}_{12}, \ldots, \mathfrak{R}_{1l}\}, \{\mathfrak{R}_{21}, \mathfrak{R}_{22}, \ldots, \mathfrak{R}_{2l}\}, \ldots, \{\mathfrak{R}_{n1}, \mathfrak{R}_{n2}, \ldots, \mathfrak{R}_{nl}\},$$

где

$$\mathfrak{R}_{ij} = \begin{cases} R_{ij}, j \le j_b, \\ \mathcal{R}_{ij-j_b}, j > j_b. \end{cases}$$

Первый случай — это базовое отношение, второй — отношение, стоящее слева в $(j - j_b)$-й инструкции.

Теорема 2. *Каждое отношение $\mathfrak{R}_{ij}$ в расширенной последовательности анализов имеет инкрементальное разложение по отношению $\mathfrak{R}_{i-1j}$.*

Доказательство. Индукция по зависимостям между отношениями внутри анализа.

База. Для базовых отношений такое разложение было построено выше — см. (12), (13).

Переход. Пусть отношения $\{\mathcal{R}_{ik} \,|\, k < j\}$ имеют инкрементальные разложения. Рассмотрим j-ю инструкцию. В общем случае она будет иметь вид

$$\mathcal{R}_{ij} := \mathcal{R}_{ip} \oplus \mathcal{R}_{iq}, \text{ причем } p, q < j. \tag{14}$$

Согласно теореме 1, каждая операция $\oplus$ имеет инкрементальное разложение, поэтому (14) можно переписать так:

$$\begin{aligned}\mathcal{R}_{ij} \quad := \quad & (\mathcal{R}_{ip}^{base} \oplus \mathcal{R}_{iq}^{base}) \cup \\ & F(\mathcal{R}_{ip}^{base}, \mathcal{R}_{ip}^{add}, \mathcal{R}_{ip}^{sub}, \mathcal{R}_{iq}^{base}, \mathcal{R}_{iq}^{add}, \mathcal{R}_{iq}^{sub}) \setminus \\ & G(\mathcal{R}_{ip}^{base}, \mathcal{R}_{ip}^{add}, \mathcal{R}_{ip}^{sub}, \mathcal{R}_{iq}^{base}, \mathcal{R}_{iq}^{add}, \mathcal{R}_{iq}^{sub}). \qquad (15)\end{aligned}$$

По индукционному предположению отношения $\mathcal{R}_{ip}$ и $\mathcal{R}_{iq}$ имеют инкрементальные разложения по отношениям $\mathcal{R}_{i-1p}$ и $\mathcal{R}_{i-1q}$ соответственно, поэтому можно записать $\mathcal{R}_{ip}^{base} = \mathcal{R}_{i-1p}$ и $\mathcal{R}_{iq}^{base} = \mathcal{R}_{i-1q}$; принимая в расчет то, что $\mathcal{R}_{i-1j} = \mathcal{R}_{i-1p} \oplus \mathcal{R}_{i-1q}$,

$$\begin{aligned}\mathcal{R}_{ij} \quad := \quad & \mathcal{R}_{i-1j} \cup \\ & F(\mathcal{R}_{ip}^{base}, \mathcal{R}_{ip}^{add}, \mathcal{R}_{ip}^{sub}, \mathcal{R}_{iq}^{base}, \mathcal{R}_{iq}^{add}, \mathcal{R}_{iq}^{sub}) \setminus \\ & G(\mathcal{R}_{ip}^{base}, \mathcal{R}_{ip}^{add}, \mathcal{R}_{ip}^{sub}, \mathcal{R}_{iq}^{base}, \mathcal{R}_{iq}^{add}, \mathcal{R}_{iq}^{sub}). \qquad (16)\end{aligned}$$

А это и будет требуемым инкрементальным разложением $\mathcal{R}_{ij}$ по $\mathcal{R}_{i-1j}$. □

2.3.3. Алгоритм инкрементального анализа

Итак, все готово для описания алгоритма инкрементального анализа последовательности данных $D_1, D_2, \ldots, D_n$.

Введем дополнительный фиктивный элемент данных D_0, для которого все отношения в его анализе будут пустыми: $R_{0j} = \varnothing$. В случае анализа кода это может быть «пустая программа».

При поступлении данных D_i строим i-й анализ. Сначала вычисляем базовые отношения (например, методом «долей», описанным выше). Затем продолжаем вычислять подряд все отношения в расширенном анализе, интерпретируя трехадресный код, который был построен для определения расширенного анализа. При этом для вычисления j-го отношения используем разложение (16), т. е. вычисляем только функции F и G, а $\mathcal{R}_{i-1j}$ берем из предыдущего анализа. В итоге самая дорогостоящая операция разложения (15) — $\mathcal{R}_{ip}^{base} \oplus \mathcal{R}_{iq}^{base}$ — будет «закэширована» на этапе предыдущего анализа.

3. Результаты

В рамках данной работы был реализован программный продукт, включающий в себя реализации библиотек для работы с BDD и отношениями, реализацию интерпретатора языка `ERML`, набор алгоритмов статического анализа кода на языке Java и плагин к среде `Eclipse` для визуализации результатов анализа.

Чтобы показать эффективность инкрементального подхода, было произведено сравнение обычного анализа указателей и его инкрементальной версии. Алгоритм анализа указателей в обоих случаях был один и тот же. Тестирование производилось на компьютере с процессором AMD Athlon XP 1700+ и 768MB оперативной памяти. Для сравнения были выбраны восемь программ на Java, взятые с *http://www.sourceforge.net*. Эти программы были выбраны абсолютно произвольно. Среди них: пакет генетических алгоритмов `JGap`, программа для работы с почтовыми протоколами SMTP и POP3 `CamelsEye`, визуализатор молекул химических соединений `sketchEl`, средство расчета тестового покрытия `cobertura` и др. Все программы имеют размер исходного кода в пределах одного мегабайта.

Первое сравнение производилось следующим образом. Сначала программа анализировалась, затем подвергалась изменению, а затем измененная программа инкрементально переанализировалась. В качестве изменений выбирались такие, которые максимально усложняют задачу пересчета.

В таблице 1 приведено сравнение времени работы полного анализа «с нуля» (столбец «время полн. анализа») и время работы инкрементального пересчета (столбец «время инкр. анализа»). В последнем столбце вычислено процентное соотношение времени пересчета к времени полного анализа.

По результатам, приведенным в таблице 1, можно сказать, что время пересчета на разных тестах разное, но есть тенденция к уменьшению относительного времени пересчета при увеличении размера тестов.

В таблице 2 приведены сведения о размере инкрементального разложения (см. раздел 2.2) отношения $PointsTo$, которое по сути является главным результатом тестируемого анализа указателей. Во втором столбце таблицы указан размер отношения $PointsTo_{base}$, которое содержит элементы, не изменившиеся по сравнению с первоначальным вариантом тестовой программы. В третьем столб-

Таблица 1. **Время работы инкрементального анализа указателей**

программа	размер, K	строк кода	время полн. анализа, сек	время инкр. анализа, сек	%
GSA	83	3224	4.7	1.4	30
sketchEl	174	5412	16.4	3.7	22
bqsource	351	11034	14.7	5.9	40
javaBDD	368	13132	18.2	7.5	41
cobertura	597	23901	39.1	2.9	7
JGap	766	24064	28.3	3.2	11
BCEL	827	27694	50.8	7.7	15
CamelsEye	844K	23819	41.8	8.3	20

це указан размер отношения $PointsTo_{add}$ — это результат добавления оператора в тестовую программу. В последнем столбце указано процентное соотношение размера измененной части отношения ($PointsTo_{add}$ и $PointsTo_{sub}$) к размеру $PointsTo_{base}$.

Таблица 2. **Характеристики инкрементального разложения отношения PointsTo**

программа	размер *base*	размер *add*	размер *sub*	%
GSA	7423	783	258	10
sketchEl	29792	482	25	1.6
bqsource	12790	1450	43	11
javaBDD	165395	37862	95	23
cobertura	69302	2422	44	3.5
JGap	20413	92	28	0.5
BCEL	107778	332	4	0.3
CamelsEye	51256	502	130	1

По результатам, приведенным в таблице 2, можно увидеть ту же тенденцию, что и в предыдущей таблице — разброс результатов и уменьшение относительного размера при увеличении размера тестов. Особо стоит отметить результат сравнения на тесте `javaBDD`, показавшем наихудшую эффективность инкрементального пересчета в таблице 1. Понять, почему так произошло, можно, взглянув в соответствующую строку в таблице 2 — действительно, изменения, которым была подвергнута тестовая программа, подействовали на множества указываемых значений 23% указателей в программе. Это очень много для одного локального изменения, и относительное время инкрементального пересчета 41% — хороший результат для него.

Если результаты первого сравнения доказали эффективность инкрементального анализа при инкрементальном пересчете, цель второго сравнения — показать, насколько ухудшились характеристики анализа при переходе от обычных отношений к отношениям, построенным на основе инкрементальных разложений. Эффективность второго варианта отношений при инкрементальном пересчете показана выше, а здесь оба варианта сравниваются при полном анализе «с нуля». Тестирование производилось на тех же примерах, но, в отличие от предыдущего сравнения, где в расчет принимался только непосредственно скрипт анализа указателей, здесь в расчет принимается полный анализ, включающий в себя построение базовых отношений методом «долей», работу скрипта анализа указателей и других скриптов, необходимых для анализа указателей.

На диаграмме 1 (рис. 1) показан результат сравнения времени работы анализа. Серые столбики диаграммы соответствуют обыч-

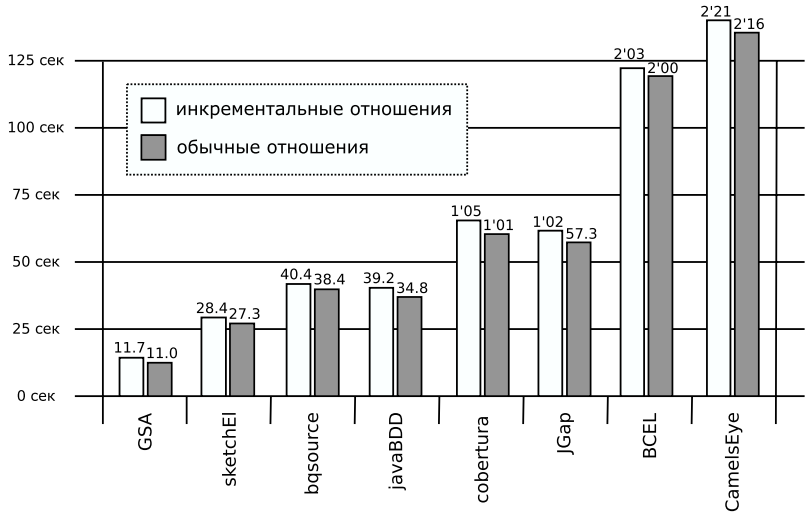

Рис. 1. Полное время работы анализа указателей

ной версии отношений, а белые — инкрементальной. На диаграмме 2 (рис. 2) показан результат аналогичного сравнения по расходу памяти.

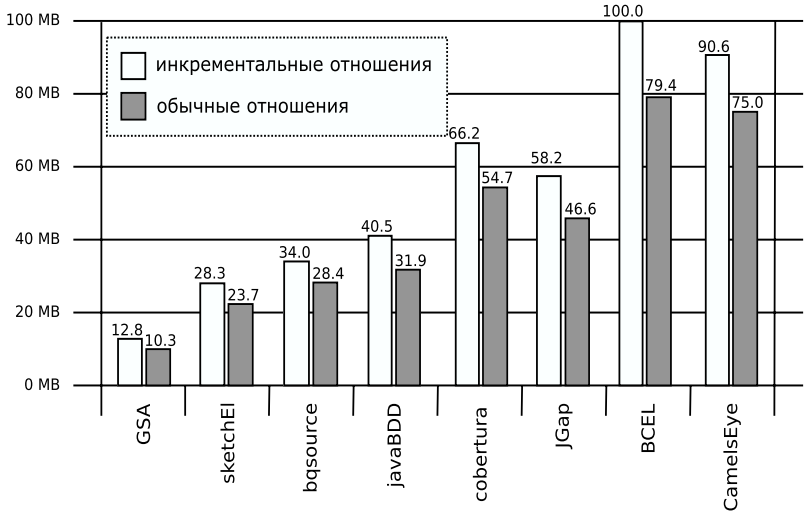

Рис. 2. Использование памяти в анализе указателей

По результатам, показанным на диаграммах (см. рис. 1, 2), можно сделать вывод о том, что переход к инкрементальному разложению отношений почти не ухудшил временны́х характеристик анализа и расход памяти (максимальное замедление — 6%, а максимальное увеличение памяти — 26%). Последний результат особенно важен, так как кажущееся увеличение расхода памяти огромно — на каждое отношение нужно хранить три дополнительных отношения для инкрементального разложения и разбиение отношения по долям, если оно является базовым (см. раздел 2.3). Здесь подтвердилась гипотеза о том, что все эти дополнительные отношения схожи по структуре, и хранение отношений в BDD будет использовать этот факт, в результате чего размер инкрементализованных отношений будет ненамного больше первоначальных.

В итоге тесты показали, что инкрементальная версия анализа практически не уступает обычной при полном анализе «с нуля», а при инкрементальном пересчете изменений она показывает заметное улучшение.

Заключение

В рамках данной работы был предложен новый подход к инкрементализации алгоритмов статического анализа кода. Подход основан на описании алгоритмов анализа в терминах отношений и операций над ними и последующем их инкрементальном разложении. Была показана высокая эффективность структуры данных BDD применительно к статическому анализу и, в особенности, инкрементальному.

В итоге был разработан готовый программный продукт, интегрированный в среду `Eclipse`, позволяющий программисту решать целый спектр задач из области понимания программ.

Список литературы

[1] *Aho A. V., Sethi R., Ullman J. D.* Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986. 796 p.

[2] *Andersen L.O.* Program Analysis and Specialization for the C Programming Language. Ph.D thesis. DIKU, University of Copenhagen. 1994. 311 p.

[3] *Atkinson D. C., Griswold W. G.* The design of whole-program analysis tools // Proc. 18th International Conference on Software Engineering. 1996. P. 16–27.

[4] *Berndl M., Lhoták O.* Points-to analysis using BDDs // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. 2003. P. 103–114.

[5] *Besson F., Jensen T.* Modular class analysis with DATALOG // 10th International Symposium on Static Analysis, LNCS 2694. 2003.

[6] *Beyer D., and Noack A.* CrocoPat 2.1 Introduction and Reference Manual. Computer Science Division (EECS), University of California, Berkeley. http://mtc.epfl.ch/~beyer/CrocoPat/.

[7] *Chamberlin D. D., Boyce R. F.* SEQUEL: A structured English query language // Proc. ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control. 1974. P. 249–264.

[8] *Klint P.* A Tutorial Introduction to RSCRIPT. http://homepages.cwi.nl/~paulk/courses/SoftwareEvolution/rscript-tutorial.pdf. 27 p.

[9] *Lam M. S., Whaley J.* Context-sensitive program analysis as database queries // Proc. ACM Symposium on Principles of Database Systems. 2005. P. 1–12.

[10] *Nielson R., Nielson H., Hankin C.* Principles of Program Analysis. Springer-Verlag, 1999. 476 p.

[11] Prechelt L., Kramer G. Functionality versus practicality: Employing existing tools for recovering structural design patterns // Journal of Universal Computer Science, 4(12). 1998. P. 866–882.

[12] *Steensgaard B.* Points-to analysis in almost linear time // ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1996. P. 32–41.

[13] *Tip F.* A Survey of Program Slicing Techniques // Journal of Programming Languages. Vol. 3. N. 3. 1995. P. 121–189.

[14] *Ullman J. D.* Principles of Database and Knowledge-Base Systems. Vol. I and II. W.H. Freeman and Company, 1989.

[15] *Vivien F., Rinard M.* Incrementalized pointer and escape analysis // Conference on Programming Language Design and Implementation. 2001. P. 35–46.

[16] *De Volder K.* JQuery: A Generic Code Browser with a Declarative Configuration Language. University of British Columbia, Vancouver, Canada. http://www.cs.ubc.ca/labs/spl/projects/jquery. 15 p.

[17] *De Volder K.* Type-Oriented Logic Meta Programming. Ph. D. Thesis, Vrije Universiteit Brussel. http://tyruba.sourceforge.net. 226 p.

[18] *Wagner T., Graham S.* Incremental Analysis of Real Programming Languages // Proc. SIGPLAN Conference on Programming Language Design and Implementation. 1997. P. 31–43.

[19] *Whaley J., Lam M.* Cloning-based context-sensitive pointer alias analyses using binary decision diagrams // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. 2004. P. 131–144.

[20] *Yur J., Ryder B. G., Landi W.* An incremental flow- and context-sensitive pointer aliasing analysis // Proc. Twenty-First International Conference on Software Engineering. 1999. P. 442–451.

[21] *Zhang X., Gupta R., Zhang Y.* Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams // IEEE/ACM International Conference on Software Engineering. 2004.

[22] *Zhu J., Calman S.* Symbolic pointer analysis revisited // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. 2004. P. 145–157.

Линеаризация графа потока управления с учётом результатов профилирования

О. В. Медведев
dours@tercom.ru

В данной работе рассматривается построение оптимальной линеаризации (линейного упорядочения вершин) графа потока управления с учётом результатов профилирования. Под профилированием в данном случае понимается подсчёт количества инструкций перехода, выполненных процессором на репрезентативном наборе тестов. Критерий оптимальности линеаризации — количество тактов, потраченное процессором на команды перехода при выполнении оптимизируемой программы на том же наборе тестов. Предложены два алгоритма — полиномиальный и перебор с отсечениями. Произведено их сравнение с известными алгоритмами по результатам линеризации нескольких стандартных программ.

Введение

Одним из важных средств ускорения выполнения инструкций в современных процессорах является конвейер (pipeline). Идея его использования заключается в разбиении процедуры выполнения команды на несколько стадий с целью их параллельного исполнения. В некоторых процессорах число стадий очень велико (например, 31 в Intel Pentium на ядре Prescott[1]). Этот метод используется не только в процессорах общего назначения, но и в DSP-процессорах.

Слабым местом такой схемы, как известно, являются условные переходы. В момент выборки инструкции условного перехода процессор не знает, произойдёт ли этот переход. Тем не менее на сле-

[1] http://en.wikipedia.org/wiki/Pentium_4.