Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Анализ указателей в среде Promise

*Дипломная работа студента 59 группы
Симановского Андрея Александровича*

Научный руководитель:	.... (Булычев Д. Ю.)
Рецензент:	.... (Друнин А. В.)
"Допустить к защите", заведующий кафедрой, профессор, д.ф-м.н.	.... (Терехов А. Н.)

*Санкт-Петербург
2002*

Оглавление

Глава 1

Введение

Анализ указателей - это метод определения в период компиляции возможных значений переменных, имеющих тип указатель, т.е., его целью является построение консервативной оценки значений всех выражений, имеющих тип указатель, в программе[1]. Наличие знаний о возможных значениях таких выражений позволяет компилятору за счет возрастающей точности других анализов более эффективно применять большинство оптимизирующих преобразований. Например, анализ указателей позволяет улучшить удаление общих подвыражений, чистку циклов, увеличивает время жизни значений и, тем самым, позволяет эффективнее распределять регистры и уменьшать количество загрузок значений из памяти.

Кроме того, возможности компилятора по распознаванию параллелизма в программе и, как следствие, генерации качественного ILP-кода[2] для VLIW[3] архитектур также находятся в зависимости от наличия эффективного анализа указателей, так как именно полученные при помощи него знания позволяют для неперекрывающихся фрагментов памяти переупорядочивать обращения к ним, что является основным действием при планировании инструкций.

Анализ указателей имеет наибольшее значение для таких активно использующих указатели языков, как C. Для последнего анализ указателей важен ещё и потому, что язык C накладывает относительно слабые типовые ограничения, что позволяет, например, хранить значение адреса некоторого объекта в переменной, имеющей произвольный тип указателя. В дальнейшем материал, изложенный в данной работе, будет касаться анализа указателей для языка C, кроме специально оговоренных случаев, а также ситуаций, когда материал может быть отнесён в равной степени как к языку C, так и к другим языкам. Стоит отметить, что существует опыт анализа указателей, выполняемого для ассемблерного кода.

[1] Существует также задача нахождения обязательных значений, но она обычно решается специфичными для каждого конкретного языка методами в front-end'е программного средства.

[2] ILP (Instruction Level Parallelism) - поддержка параллельного исполнения машинных инструкций на уровне системы команд процессора.

[3] VLIW (Very Long Instruction Word) - архитектура процессора, в которой машинная инструкция имеет большую длинну (в том числе, её размер больше размерности шины, считывающей инструкции из памяти), так как её формат содержит управляющие команды для ряда однотипных функциональных блоков процессора.

Применение анализа указателей не ограничивается сбором фактов для оптимизирующих преобразований, анализ указателей может быть также применен в анализирующих средствах разработки, например, для нахождения ряда ошибок, связанных с разыменованием некорректного указателя (выход за пределы массива, разыменование неинициализированного указателя и т.д.).

Частично некоторые задачи анализа указателей решались ещё в оптимизирующих трансляторах фирмы IBM с языков COBOL и PL/1, но серьезное внимание проблеме было уделено только в новых трансляторах для C/C++ (например, компилятор GNU C [2]) и Java. Также анализ указателей активно разрабатывался для таких языков как LISP и Fortran-90.

Прежде чем переходить к обзору существующих решений, будет уместным дать краткий обзор терминологии. Большинство работ, связанных с темой анализа указателей, датируются последними двумя десятилетиями. Авторы работ пользуются устоявшейся классификацией возможных решений задачи анализа указателей. Решения различают межпроцедурные и внутрипроцедурные. Подобная классификация связана с тем, что на уровне процедуры структура программы представлена довольно четко, обычно в виде уграфа (графа потока управления, CFG)[1], что позволяет строить на уровне процедуры как потоко-независимый анализ (т.е. анализ, не использующий информацию уграфа о порядке следования операторов в процедуре), так и потоко-зависимый (использующий уграф). Обычно решение задачи анализа указателей в программе предлагает либо потоко-зависимый, либо потоко-независимый анализ внутри процедуры, и некоторый межпроцедурный алгоритм, позволяющий объединить (и, возможно, уточнить) данные внутрипроцедурного анализа для каждой из процедур. Межпроцедурный алгоритм чаще всего использует граф вызовов процедур. В зависимости от того, различны ли действия внутрипроцедурного алгоритма для различных точек (контекстов) вызова процедур, или нет, различают контекстозависимый и контексто-независимый анализы.

Результатом работы алгоритма анализа указателей является набор фактов points-to ("указывает-на"), который в дальнейшем используется другими анализами. Следует отметить, что независимо от того, представляет ли анализ указателей множество фактов points-to как само множество, или как его дополнение до некоторой консервативной оценки, объем данного множества может расти как при уточнении, так и при загрублении анализа, поэтому эффективное представление множетсва фактов points-to, а также других результатов работы алгоритма, играет важную роль.

Другим возможным представлением мог бы быть набор фактов наличия синонимов, т.е., непустого пересечения множеств points-to для двух произвольных адресных выражений (что является более удобным с точки зрения использования), но, в силу устройства существующих алгоритмов и легкости перехода от упомянутой выше формы в виде набора множеств из points-to фактов к последнему представлению, points-to форма используется чаще.

Данная работа предлагает решение внутрипроцедурной задачи анализа указателей для языка C, реализованное в для среды Promise[2].

[1] Определение уграфа (CFG) можно найти, например, в работе [1].

[2] Подробнее о среде Promise пойдёт речь в Главе 4

В заключение к этому краткому обзору хотелось бы отметить, что даже потоко-независимый точный (в смысле существования уграфа и пути в нем, на котором реализуется совпадение значений двух данных адресных выражений в данном операторе) анализ указателей (при условии, что допустим сколь угодно большой уровень косвенности для адресных выражений) является NP-трудной задачей [3]. В случае потоко-зависимого анализа, поиск точного решения остается NP-трудным при отсуствии динамического выделения памяти, и задача становится неразрешимой при его наличии [4]. В связи с последним, стоит отметить, что предлагаемое решение есть некоторое загрубление точного анализа, о характере загрубления пойдет речь в Главе 3.

Глава 2

Обзор существующих подходов и реализаций

Как отмечено выше, существует устоявшаяся классификация алгоритмов анализа указателей. Она естественным образом возникает из-за разделения исходного текста программы на процедуры. Взаимодействие внутрипроцедурной и межпроцедурной частей алгоритма определяет качество проводимого анализа. В целом, большинство алгоритмов работают по следующей схеме:

Алгоритм 1. *Общая схема реализации анализа указателей.*
Вход: Исходная программа
Выход: Информация о синонимах
Метод:

```
foreach (Procedure p) do
   IN(p) <- <информация о статических синонимах>;
   OUT(p) <- <некоторая оценка>;
   // (в качестве оценки могут служить, например,
   // результат проведения внутрипроцедурной оценки
   // в консервативных предположениях).
done;
do
   foreach (Procedure p) do
     // (перебирая предпочтительно в топологическом
     // или ином порядке,
     // задаваемом на графе вызовов программы, PCG),

     foreach (Callsite c of p) do
       Вычислить IN(c);
       Произвести внутрипроцедурный анализ
         вызываемой процедуры,
         на основе которого вычислить OUT(c).
     done
   done
until <сойдутся IN и OUT для всех p >;
```

Приведенная схема является достаточно условной, она не специфицирует как именно происходит внутрипроцедурный анализ, она не характеризует изменение графа вызовов процедур, которое может присутствовать в случае контексто-зависимого анализа или наличия вызовов по указателю.

Данная схема может быть обоснована в рамках формализма решения задачи анализа потоков данных (здесь и далее объекты, возникающие в постановке задачи анализа потоков, используются без дополнительных пояснений; постановка общей задачи доступна в [1]). Рассмотрим алгебраическую полурешетку, элементами которой являются наборы множеств фактов points-to для всех адресующих выражений программы (множество E), допустимых в контексте данный точки вызова. Определим операцию $\wedge$ как пересечение множеств фактов points-to для всех элементов E. Определим функцию перехода для исходящей дуги вершины графа PCG как результат внутрипроцедурного анализа. Монотонность функции следует из уточняющего характера анализа. Конечная высота полурешетки обеспечивается конечностью языка описания фактов points-to (это должно быть обеспечено алгоритмом внутрипроцедурного анализа и структурой представления фактов). Данные условия позволяют утверждать, что итеративный алгоритм решения возможен, т.е. указанное в цикле алгоритма условие о сходимости информации о синонимах выполнится, и цикл остановится, выполнив конечное число шагов.

2.1 Потоко-независимый и потоко-зависимый анализы

Сначала обратимся к реализации внутрипроцедурного алгоритма. Здесь присутствуют два принципиально различных варианта.

2.1.1 Потоко-независимый анализ

Потоко-независимый анализ рассматривает операторы процедуры, влияющие на информацию о синонимах, как выполняющиеся в произвольной последовательности и выполняющиеся произвольное число раз (это отвечает ситуации, когда при последовательном исполнении все подобные операторы располагались бы в различных ветвях условного оператора, который, в свою очередь, располагался бы внутри цикла, исполняющегося неопределенное число раз). Несмотря на кажущуюся масштабность загрублений, потоко-независимые анализы дают неплохие результаты по сравнению с потоко-зависимыми, при этом значительно выигрывая в скорости, что показано в работе [5].

Потоко-независимые алгоритмы обсуждаются в работе Андресена [6], где представлен алгоритм со сложностью $O(n^3)$ (фактически, со сложностью построения транзитивного замыкания), где n — число операций, затрагивающих указатели в процедуре, в работе Штеенсгаарда [7], где представлен почти линейный (при условии использования для хранения классов эквивалентости структур link-eval, сложность есть $O(n * \alpha(n,n))$[1]) алгоритм.

[1]Где α — обратная к функции Аккермана; определение данной функции доступно в книге [1].

Последний дает больший объем итоговых множеств points-to, чем первый. Алгоритм Штеенсгаарда интересен тем, что может быть интерпретирован как задача нахождения минимальной системы типов (классов синонимов), определяемой правилами нескольких фиксированных видов (несколько видов операторов присваивания и вызовов функций), в которую могут быть вложены выражения программы; т.е., решение Штеенсгаарда в некотором роде минимальное полное решение задачи анализа указателей. За подробностями опять можно обратиться к [7].

В работе Шапиро и Хорвитц [8] приведены два алгоритма, первый из которых параметризуемый и в двух предельных случаях дает алгоритмы Андерсена и Штеенсгаарда, второй рекурсивно вызывает первый с различными параметрами, основываясь на данных анализа программы. Первый из этих двух алгоритмов будет уместно привести, так как он дает представление и о других упомянутых выше алгоритмах.

Ключевым для понимания алгоритма является понятие points-to графа.

Определение 1. *Points-to граф* — это ориентированный граф, вершинами которого являются классы эквивалентности выражений программы (выбор отношения эквивалентности, по которому строятся классы — произвол алгоритма, семантический смысл класса эквивалентности — набор неразличимых с точки зрения алгоритма обращений к памяти, т.е., класс эквивалентности — описание абстрактной ячейки памяти); дугами e соединяются те и только те классы эквивалентности, для которых верно, что существует представитель класса $beg(e)$, разыменование которого является возможным синонимом некоторого представителя класса $end(e)$.

Множества points-to для некоторого адресного выражения определяются на основе такого представления как тот или те классы эквивалентности, которые являются непосредственными потомками (как вершины графа) класса эквивалентности, представителем которого является данное адресное выражение, т.е.:

$$pointsTo(e) = \bigcup_{\overline{x} \in children(\overline{e})} \{x \in \overline{x}\}$$

где $\overline{x}$ — класс эквивалентности, представителем которого является x.

Алгоритм заключается в том, что выполняется обход всех операторов процедуры, использующих адресные выражения, и производится добавление одной или нескольких дуг, которые индуцируются данным оператором. Параметром алгоритма является допустимая степень вершины points-to графа. Для алгоритма Андресена ограничения нет, и каждое адресное выражение помещается в собственный класс эквивалентности; для алгоритма Штеенсгаарда степень вершины ограничена единицей, и, следовательно, при использовании алгоритма Штеенсгаарда, на каждом шаге добавляется только одна дуга в points-to граф.

Заметим, что последнее, строго говоря, относится только к случаю, когда не рассматриваются следствия наличия синонимов. Представим, что присваивание `p=& x` привело к образованию пары синонимов `(*p,x)`, и x является корректным указателем куда-либо. В таком случае возникают последствия — пары синонимов `(**p,*x)`. Если `*x`, в свою очередь, также является корректным указателем, то возникает пара `(***p,**x)`, и т.д. С точки зрения изложенного ранее формализма, данная ситуация представляет собой случай, когда высота полурешётки становится неограниченной. Это одна из проблем потоко-независимых алгоритмов (при потоко-зависимом анализе можно уйти от множественных разыменований путем введения временных переменных; для потоко-независимого анализа данный подход некорректен, т.к. при предположении произвольного порядка выполнения операторов, введение операций с временными переменными сильно загрубит анализ). Фактически, проблема происходит из потенциальной неограниченности данных, используемых в программе. Для аппроксимирования данных программы некоторым конечным множеством, чей объём мажорируется размером входных данных алгоритма, используется либо метод, называемый k-ограничением, сводящийся, по сути, к тому, что все адресные выражения, содержащие более, чем k операций, автоматически считаются находящимися в отношении синонимичности со всеми, либо конечность данных обеспечивается каким-либо другим образом, например, склеиванием значений, создаваемых в программе динамически (т.е., ограничением количества возникающих при анализе указателей классов эквивалентности адресных выражений). Другой, смежной проблемой, является возникновение циклов в points-to графе. Поиск циклов обычно также ограничивают рассмотрением циклов длины не более k. Данные две проблемы были исследованы Джонсом и Мучником [9]. Интересное решение проблемы анализа указателей для циклических структур предложил Дойч [10], подробнее об этом решении пойдет речь в Части 2.1.2.

2.1.2 Потоко-зависимый анализ

В отличие от потоко-независимого анализа, *потоко-зависимый анализ* учитывает порядок исполнения операторов в программе. Внутрипроцедурный потоко-зависимый анализ работает, чаще всего, на так называемом, Sparse Evaluation Graph (SEG).

Определение 2. *SEG* — это подграф уграфа процедуры, вершины которого содержат либо операторы, модифицирующие отношения синонимичности, либо представляют собой вершины, в которых происходит объединение информации о синонимах, либо являются вершинами start или stop; дуги представляют пути в исходном графе, соединяющие соответствующие вершины.

Легко заметить, что SEG является уграфом. Данный факт также позволяет формализовать решение задачи нахождения отношения синонимичности как решение задачи анализа потоков данных итеративным алгоритмом. Для каждого узла строятся множества фактов IN и OUT. Начальным решением является:

$IN(start) = \{$ отношение синонимов при входе в процедуру $\}$

для остальных узлов IN и OUT пусты.

Функцией перехода служит:

- для вершин-модификаторов отношения — результат модификации (обычно в нем выделяют генерирующую и уничтожающую части),

- для вершин, в которых сходятся результаты нескольких потоков управления,

$$IN(X) = \bigcup_{Y \in \ Pred(X)} OUT(Y)$$

Можно показать, что построенная таким образом схема при выполнении определённых условий сходится.

Отношение синонимичности также может быть представлено в виде points-to графа, как это, например, сделано Шуа и Бёрком [11], которые использовали описанный выше подход в алгоритме потоко-зависимого анализа. Эти авторы использовали транзитивную редукцию графа, чтобы избежать разрастания информации о синонимах при применении функций перехода в вершинах-модификаторах.

Интересная полурешетка предложена Дойчем [10]. Интерес представляет её параметризация численной решеткой (в качестве которой могут выступать, например, арифметические интервалы) с оператором, обрывающим бесконечные цепочки. Подход использует понятие символьного пути доступа (это, в первом приближении, последовательность операторов доступа, применяемых к некоторой переменной). Отношение синонимичности вводится для нормализованных путей доступа и элементов численной решетки, которые представляют собой количества применений отдельных операций в них. Операция `join` при этом определяется следующим образом:

Алгоритм 2. *Реализация операции join для параметризованной полурешётки.*
Вход: отношения синонимичности из параметризованнной полурешётки e_1, e_2.
Выход: $join(e_1, e_2)$.
Метод:

```
    e <- e1 ∪ e2;
    Повторять
      Если (<f1,f2>, K) и (<f1,f2>, K') лежат в e,
        //(K -- элемент численной решётки, <f1,f2> -- пара
синонимов)
        заменить их на (<f1,f2>, K ∪ K')
    Пока e не сойдется.
```

Удачный выбор численной решётки позволяет получать для циклических структур неограниченного объема невырожденные отношения синонимичности. Для ограничения высоты полурешетки используется алгоритм, нормализующий пути доступа. Он принимает на вход произвольный путь доступа и возвращает его нормализованное представление, создавая временные переменные и сворачивая последовательное применение операций в итеративные конструкции и уравнения на количества итераций.

Вид функций перехода и обоснование подхода можно почерпнуть из работы Дойча [10].

2.2 Межпроцедурный анализ

2.2.1 Зависимость от контекстов вызовов

Аналогом потоковой зависимости/независимости во внутрипроцедурном случае для межпроцедурного случая может служить учет анализом различных контекстов вызовов. Межпроцедурный анализ часто использует Procedure Call Graph (PCG).

Определение 1. *Граф вызовов процедур (PCG)* — это граф, вершины которого представляют собой функции, а дуги — наличие вызовов функций.

Учет контекста вызовов может выполняться как клонированием процедур, так и другими методами, например, расширением фактов путем добавления информации о точке вызова в элемент полурешетки (в данном случае возникает проблема аппроксимации потенциально бесконечного стека вызовов рекурсивных функций). В качетве примера клонирования процедур можно рассмотреть использование Invocation Graph, который, при отсутствии рекурсивных вызовов, дает точное отображение передачи управления между процедурами программы. В данном графе вызовов путь от main до текущей вершины представляет собой содержимое стека, которое возникнет при исполнении программы. Т.е., отличие Invocation Graph от PCG состоит в том, что в первом процедуры размножены так, что на каждый нерекурсивный вызов приходится один уникальный экземпляр процедуры.

Контексто-независимый метод делает пессимистическое предположение, объединяя отношения синонимичности, получаемые от различных точек вызова.

2.2.2 Вычисление фактов. Множества Gen и Kill

Основной трудностью является то, что потоки данных в точках вызова двунаправлены (присутствует как поток входных фактов процедуры, строящихся на основе фактов в точках вызова, так и обратный поток, обновление отношения синонимичности по выходе из процедуры). Данный факт отрицательно сказывается на качестве итеративных алгоритмов, которые применяются для решения данной задачи.

Несколько отличное от работы над PCG решение было предложено Ланди и Райдер [12]. Эти авторы предложили использовать общий для всей программы уграф, т.н., ICFG (Interprocedural CFG). Также они формулировали задачу анализа потоков данных для множества реализуемых путей (т.е. путей, на которых вызванная функция возвращает управление вызвавшей):

$$TASK(n) = \bigwedge_{p \in \ RP(n_0, n)} f_P(N, Q),$$

где N — решение для дна решетки. Здесь представлено решение, имеющее ещё один параметр, это связано с тем, что рассматриваемая полурешетка усложнена наличием дополнительного множества (E_+), которое отражает изменение ICFG по мере уточнения информации о вызовах по указателю. Данное расширение не содержалось в работе Ланди и Райдер [12], которые рассматривали только малое подмножество языка C. Формулировка взята из другой работы, использующей аналогичный подход.

Для выражения результата уточнения множества фактов в точке вызова, происходящего в результате использования данных внутрипроцедурного анализа для вызываемой процедуры, удобно последние представлять в виде двух множеств, называемых Gen и Kill.

Определение 2. *Множества Gen и Kill* — это множества, представляющие собой результаты работы внутрипроцедурного анализа. Множество Gen является множеством отношений синонимичности, устанавливаемых процедурой, множество Kill является множеством отношений синонимичности, уничтожаемых данной процедурой. Тогда множество фактов OUT для точки вызова может быть определено как

$$OUT(c) = IN(c) \cup Gen(p[,c]) \setminus Kill(p[,c])$$

где p — вызываемая процедура, c — точка вызова $Gen()$ и $Kill()$ — функции, возвращающие множества Gen и Kill соответственно для процедуры p, и, возможно, зависящие от точки вызова c.

При межпроцедурном анализе в точке вызова процедуры как методы, использующие потоко-зависимый, так и использующие потоко-независимый внутрипроцедурный анализ, могут использовать уточнение множеств Gen и Kill. Для этого удобны предвычисляемые множества *ForwardGen*, *BackwardGen* и *ForwardKill*, *BackwardKill*, которые определяются как соответствующие добавляемые (удаляемые) на любом пути от start до точки вызова (ForwardGen, ForwardKill) или от точки вызова до stop (BackwardGen, BackwardKill) отношения синонимичности. Таким образом можно упростить внутрипроцедурные перевычисления при итеративном вычислении межпроцедурных фактов. Данный подход был реализован в работе Хайнда, Бёрка и др. [13].

2.2.3 Проблемы построения полурешетки

В упомянутых решениях использовался итеративный алгоритм для нахождения межпроцедурных фактов. Здесь будет уместным отметить трудности (помимо двунаправленности потоков данных), с которыми неизбежно приходится сталкиваться при разработке межпроцедурного анализа:

- возможная недистрибутивность функций перехода (ухудшает временные характеристики и может загрубить точность),
- модифицируемость PCG в результате работы алгоритма (необходимо включать сходимость PCG в условия завершения работы алгоритма),
- наличие нереализуемых путей в PCG (движение информации о синонимах по нереализуемым путям загрубляет анализ).

Дистрибутивность функций не является обязательным условием, но желательна для более более быстрой сходимости итеративного алгоритма. Отсутствие дистрибутивности функций перехода, в основном, связано с наличием модифицируемого PCG, так как это, в свою очередь, вызывает усложнение полурешетки. Влияние на полурешетку также оказывает наличие чувствительности к контекстам вызовов, если оно реализуется через введение в том или ином виде стека вызовов как части элемента полурешетки. В таком случае важна конечность высоты полурешетки. Дистрибутивность можно обеспечить, изолируя факты, относящиеся к различным точкам вызова.

Проблема модифицируемости PCG возникает при наличии вызовов через указатели. Обычно алгоритмы применяют первоначально оптимистическое построение PCG (т.е., изначально полагается, что вызовы по указателю ничего не вызывают, и только первая итерация внутрипроцедурного анализа, устанавливающая возможные значения указателя, создает множественный вызов), это делается, чтобы не анализировать недостижимые процедуры. Предлагаются различные методы учета модифицируемости. Они сводятся к дополнению решетки информацией об указателях на функции или вызовах по ним. Одно из решений было упомянуто ранее: для ICFG вводится множество дуг $E+$, представляющих собой вызовы по указателю. В также упоминавшейся работе Хайнда, Бёрка и др. [13] использовались конетексто-зависимые синонимы для функций. Ещё одно решение — это частично отложенное вычисление функций перехода (Вильсон [14]).

Поток фактов через нереализуемые пути может, на первый взгляд, показаться основной проблемой, однако работа Шапиро и Хорвитц [8] показывает, что данная неточность многих анализов оказывает не столь сильное влияние на качество конечного результата, как можно было бы ожидать.

Завершая обзор, можно сказать, что методы анализа указателей активно развиваются, есть определенные достижения, но существуют условия, при которых в настоящее время нет вполне удовлетворительных решений задачи:

- наличие циклических структур данных
- наличие рекурсивных вызовов функций

Также, в настоящее время реализации межпроцедурного анализа неактивно используют выгоды внутпripроцедурного потоко-зависимого анализа.

Кроме того, существует ряд проблем, выходящих за рамки данной работы, которые могут быть темой дальнейшего исследования, например:

- сравнительный анализ различных реализаций внутрипроцедурного анализа и степени их приближения к точному (в смысле, указанном в Главе 1) решению
- сравнительный анализ различных реализаций межпроцедурного анализа

Глава 3

Описание предлагаемого подхода

В данной главе излагается предлагаемый алгоритм анализа указателей, теоретическое обоснование и основные характеристики алгоритма, преимущества и недостатки решения. Структура изложения решения следующая:

- Описание структуры алгоритма
- Описание контекстов
- Описание внутрипроцедурного алгоритма
- Параметризуемость алгоритма, сложность алгоритма
- Преодоление в рамках алгоритма стандартных трудностей

3.1 Описание структуры алгоритма

Предлагаемый алгоритм проводит внутрипроцедурный потоко-зависимый анализ указателей. Ниже приведены интерфейсы входа и выхода алгоритма.

Входными данными для алгоритма являются уграф процедуры, операторная разметка уграфа процедуры, данные о статических синонимах и данные о начальных значениях выражений. Рассмотрим каждое данное более подробно.

Уграф, как отмечено выше, определяется в соответствии с [1].

Определение 1. *Операторная разметка* — это отображение вершин уграфа в множество операторов, которые могут быть:

- либо присваиванием выражений
- либо тестом выражения
- либо стопом от набора выражений

где выражения есть множество термов, строимое на основе счетного множества примитивов доступа, конечного множества операций с определенной арностью, конечного множества типов, и имеющее отображение из себя в множество типов.

Данные о статических синонимах представим, как отбражение множества термов в себя:

Определение 2. *Статические синонимы* — это отображение s множества термов E в себя, обладающее свойствами:

- $\forall op \in Ops\ \ \forall e_1^s \in s(e_1) ... \forall e_n^s \in s(e_n)\ \ op(e_1^s, .., e_n^s) \in s(op(e_1, .., e_n))$
- $\forall e^s \in s(e)\ typeof(e^s) = typeof(e)$
- $\forall e^s \in s(e)\ s(e^s) = s(e)$

где Ops — множество операций, $typeof$ — тип выражения (отображение из E в T — множество типов), $e, e_0, e_1, ..$ — произвольные выражения.

Определение 3. *Начальные значения* — это отображение множества термов E в множество $E \cup Inputs$ (где $Inputs$ — некоторое счётное множество), обладающее свойствами, аналогичными свойствам отображения s.

Выходом алгоритма является отображение, которое по данному выражению и узлу уграфа выдает пару $(before, after)$, соответствующую синонимам данного выражения в программе до и после исполнения оператора, находящегося в данном узле.

Определение 4. *Потоко-зависимая информация о синонимах* — это пара отображений $before, after : E \times V_G \to E^*$, обладающих свойствами:

- $\forall e\ e \in aliases(e)$
- $\forall e\ \in aliases(e_0)\ aliases(e) = aliases(e_0)$

где вместо $aliases$ следует согласованно подставить $before$ или $after$.

Описание метода работы алгоритма излагается ниже. Отметим, что оно состоит из двух частей. Первая часть решает задачу построения суперсостояния, суть которой будет изложена в Части 3.2. Вторая часть, которую также можно назвать клиентской, извлекает из решения первой части информацию о синонимах в том объеме, который необходим для данного преобразования.

3.2 Описание контекстов

Прежде, чем описывать внутрипроцедурный алгоритм, необходимо представить ряд определений и фактов, на которые он опирается.

Пусть дана схема программы, заданная в виде пары, состоящей из уграфа G и операторной разметки (определения даны в Части 3.1).

Определим *программу* π как пару, состоящую из схемы и интерпретации множества операций.

Теперь введем определения состояния и среза как нижеследующие.

Определение 1. *Состояние* σ — это отображение

$$\sigma : \mathbb{N} \cup \{\perp\} \to \mathbb{B} \cup \perp$$

(где $\mathbb{B} = 0, 1$; причем $\sigma(\perp) = \perp$), т.е., память программы.

Будем обозначать $\pi(\sigma_0)$ *траекторию программы* с начальным состоянием памяти σ_0 — последовательность пар из узла уграфа и состояния памяти или символов `stop` или `error`, начинающуюся парой $(start_G, \sigma_0)$. Траектория формируется в результате интерпретации схемы программы с вычислением переходов и новых состояний, использующим интерпретацию программой множества операций, т.е., представляет собой изображение работы программы с памятью.

Определение 2. *Срез программы* $\pi|_v$ *(где v — оператор)* — определим формулой

$$\pi|_v = \bigcup_{\forall \sigma_0} \{\sigma : (v, \sigma) \in \pi(\sigma_0)\}$$

Можно рассмотреть задачу нахождения *инвариантного состояния*:

$$inv_\pi^v = \bigwedge_{\sigma \in \pi|_v} \sigma \quad (*)$$

где пересечение отображений σ из некоторого множества A определяется как

$$\bigwedge_{\sigma \in A} \sigma(i) = \begin{cases} b, & \forall \sigma \in A \ \ \sigma(i) = b, \ b \neq \perp \\ \perp, & otherwise \end{cases}$$

Назовем *суперсостоянием* отображение, определяемое формулой

$$\forall i \in \mathbb{N}_0 \ \ super_v^\pi(i) = \begin{cases} \sigma(i) : \sigma \in \pi|_v, & \forall \sigma \in \pi|_v \sigma(i) \neq \perp \\ \perp, & \exists \sigma \in \pi|_v \sigma(i) = \perp \end{cases}$$

Загрублением задачи нахождения суперсостояния назовем нахождение отображения $super\prime_v^\pi$, удовлетворяющего свойству:

$$\forall i \ super_v^\pi(i) \subset super\prime_v^\pi(i) \quad (**)$$

Пару решений задач (*) и (**) для данного оператора назовём *контекстом*[1]. Заметим, что нахождение суперсостояния есть решение задачи нахождения возможных значений памяти, а, следовательно, включает решение задачи анализа указателей как задачу поиска возможных значений выражений программы, имеющих тип указатель.

Все дальнейшие обоснования и результаты проводятся и получены в предположении существования алгоритма решения задачи (*). Данная работа не доказывает существование подобного алгоритма. Практическая реализация предлагаемого в работе алгоритма использует реализацию метода решения этой задачи. В предположении, что существует решение задачи (*), обоснуем возможность получения решения задачи (**). Очевидно, что, если $inv_\pi^v(i) = \alpha$ (и $\alpha \neq \perp$), то $super_\pi^v(i) = \{\alpha\}$. Обоснуем следующий факт:

[1] Не следует путать данное понятие контекста с контекстом вызова процедуры.

Утверждение 1. Для программы π и её произвольного оператора v верно, что, если для всех вершин, являющихся началами дуг из $in(v)$, решены задачи (*) и (**), и для v решена задача (*), то для этого оператора можно построить решение задачи (**).

Для данного оператора v для i таких что $inv_\pi^v(i) = \alpha(\alpha \neq\perp)$, вычисление $super_\pi^v(i)$ тривиально. Рассмотрим остальные i. Не умаляя общности, можно считать, что оператор является либо оператором присваивания, таким что $\exists i \; \sigma_k(i) \neq \sigma_{k+1}(i)$, где σ_k — значение памяти до присваивания, σ_{k+1} — значение памяти после присваивания (оператором, затрагивающим значение $super_\pi^v(i)$), либо оператором, не влияющим на значение $super_\pi^v(i)$. Построим объединение конечного числа решений задачи (**) по входящим дугам:

$$s\prime = \bigcup_{v\prime = in(v)} super_\pi^{v\prime}$$

Если оператор v не является оператором, затрагивающим значение $super_\pi^v(i)$, то $super_\pi^v(i) = s\prime$. Иначе

$$super_\pi^v(i) = \begin{cases} \beta, & inv_\pi^v(i) \neq\perp \\ s\prime(i) \cup \beta, & \perp\notin s\prime(i) \cup \beta \\ \perp, & \perp\in s\prime(i) \cup \beta \end{cases}$$

где β — значение решения задачи (**) для правой части оператора присваивания, построенное аналогично $s\prime$(i).

Соответствие таким образом построенного $super\prime_v^\pi$ решению задачи (**) можно показать, доказав включение $super_\pi^v(i)$ в $super\prime_\pi^v(i)$. Возьмем значение τ, принадлежащее построенному решению. Если,

- оператор не затрагивает значение, то, беря σ_k такое, что $\sigma_k(i) = \tau$, получим $\sigma_{k+1}(i) = \tau$
- оператор затрагивает значение, то, беря σ_k такое, что $\sigma_{k+1}(j) = \tau$, где j — номер правой части присваивания, получаем необходимое $\sigma_{k+1}(j) = \tau$

Таким образом, доказано включение *super* в *super$\prime$*, откуда следует справедливость утверждения.

Данное рассуждение позволяет реализовать алгоритм нахождения суперсостояний как часть алгоритма нахождения инвариантных состояний, о чем пойдет речь в Части 3.3.

3.3 Описание внутрипроцедурного алгоритма

3.3.1 Применение контекстов

Теперь можно перейти непосредственно к описанию логики работы внутрипроцедурного алгоритма. Предлагаемый алгоритм строит загрубление решения задачи нахождения суперсостояний в условиях, изложенных в предыдущей части. Алгоритм использует действия, аналогичные изложенным в обосновании утверждения, сформулированного в предыдущей части.

На каждом шаге алгоритма выполняется построение $super\prime^{\pi}_{v}$, следующим образом. Производится параллельное решение задач нахождения инвариантного состояния и загрубления суперсостояния. На каждом шаге сначала строится контекст перед выполнением оператора. Если мощность множества $in(v)$, где v — данный оператор, больше единицы, то решение задачи (*) перед построением контекста будет содержать неопределенные значения для тех номеров i, которым в различных предшественниках i соответствовали неодинаковые значения $inv^{\pi}_{v\prime}(i)$. Для таких узлов положим

$$super^{\pi}_{v}(i) = \bigcup_{v\prime = beg(e) | e \in in(v)} super^{\pi}_{v\prime}$$

Если данный оператор является присваиванием, то для всех зависимых выражений с соответствующими номерами i $super\prime^{\pi}_{v}(i)$ дополняется множеством $super\prime^{\pi}_{v_0}(j)$, полученным на предыдущем шаге (оператор v_0), где j соответствует правой части оператора присваивания.

3.3.2 Функция distance

Стоит отметить модификацию алгоритма построения решения задачи нахождения инвариантного состояния, которая была произведена в связи с требованиями структуры использованного метода решения задачи нахождения инвариантного состояния. Для представления отображений σ используется частичная таблица их значений. Согласно введенной формализации, отображения σ принимают номера битов памяти программы. Непосредственные номера битов памяти не могут быть вычислены при работе с внутренним представлением. Для записи аргументов отображения σ используется представление номеров битов в виде дизъюнктных множеств битов, *компонент*. Для выражений программы рассмотрим функцию $\alpha^{v}_{I} : E \times T \to \mathbb{N}^*$ (где T — множество типов, E — множество выражений) которая в данной интерпретации устанавливает соответствие между выражениями и номерами битов в памяти в контексте данного оператора v (когда оператор v понятен из контекста, будем писать просто α_I). Допустим, что разбиение на компоненты для данного оператора v и любого выражения e с типом t обладает свойством

$$\exists c \in Components \;\; \forall I \;\alpha^{v}_{I}(e,t) \in c$$

Назовем такое разбиение на компоненты *корректным.* Определим функцию $distance : (E \times T) \times (E \times T) \to \mathbb{N}$ как

$$distance(x,y) = \begin{cases} 0, & \alpha_I(x) \cap \alpha_I(y) \neq \emptyset \\ |i-j|, & \exists c \in Components \;\; \alpha_I(x) \in c, \alpha_I(y) \in c \\ \infty, & \alpha_I(x) \in c, \alpha_I(y) \in c\prime, c \neq c\prime \\ \perp, & otherwise \end{cases}$$

При наличии корректного разбиения вычисление $distance$ позволяет, не вычисляя значение α_I для данного выражения непосредственно, определять принадлежность его той или иной компоненте. Это можно осуществить, перебирая выражения и вычисляя значение $distance$ между ними и данным.

Таким образом, если описанные выше решения задач нахождения инвариантного и суперсостояния решать с точностью до значений компонент, т.е., сузить пресечение следующим образом

$$\widetilde{\bigwedge_{\sigma\in\pi|_v}} \sigma(i) = \begin{cases} \bigwedge_{\sigma\in\pi|_v}\sigma(i), & \forall j \in c_i \ \ \bigwedge_{\sigma\in\pi|_v}\sigma(j) \neq\perp \\ \perp, & \exists j \in c_i \ \ \bigwedge_{\sigma\in\pi|_v}\sigma(j) =\perp \end{cases}$$

то для описания σ достаточно описать множество компонент и функцию $distance$.

Функция $distance$, как видно из её определения, зависит от контекста оператора v. Построение решения задачи нахождения суперсостояний уточняет этот контекст, тем самым, желательным условием является энергичность вычисления функции $distance$ (т.е., её результат должен быть доступен в вычисленном состоянии при дальнейшей работе с конекстом). Это позволяет использовать данные алгоритма построения суперсостояний для уточнения разбиения на компоненты.

Функцию $distance$ можно реализовать как алгоритм вычисления разности значений синтезируемого атрибута, являющихся адресами данных в абстрактной памяти, для деревьев, представляющих собой термы. Определим атрибут $offset$ как терм над алфавитом значений адресов выражений, которые не определены без выбора интерпретации. Легко видеть, что значения этого атрибута — элементы множества выражений программы. Разность термов определим согласно следующей формуле:

$$diff(e1, e2) = \begin{cases} x, & \forall I \ \ \alpha_I(e1) - \alpha_I(e2) = x \\ \perp, & otherwise \end{cases}$$

Точное вычисление $diff$ является неразрешимой задачей, но можно построить приближенное значение $diff$, сравнивая нормализованные представления термов на синтаксическую эквивалентность.

Назовем *нормализованным представлением терма* терм e_0, получаемый из данного применением последовательности перестановочных правил для операций, содержащихся в данном терме, и обладающий следующим свойством

$$\forall \ \ w(e_0) \leq w(e)$$

где $w(e)$ — весовая функция вершины со свойствами:

- вес константы не больше веса любого узла
- вес уникального значения пропорционален его номеру
- вес операции больше веса примитивного терма
- вес операции тем больше, чем больше её приоритет

Функцию $distance$ можно реализовать как построение нормализованного представления операндов и последующее сравнение деревьев. Данное решение было использовано при реализации энергичной функции $distance$.

3.3.3 Клиентская часть алгоритма

Перейдем к описанию клиентской части алгоритма. Клиентская часть строит на основе найденных возможных значений памяти программы в каждом узле уграфа информацию о синонимах, т.е., строит пару отображений $before$ и $after$. В простейшем случае это может быть выполнено следующим алгоритмом.

Алгоритм 3. *Общая схема реализации анализа указателей.*
Вход: Уграф, отображения σ_v, разбиения $Components_v$, рабочие множества E_v
Выход: Отображения $before$ и $after$
Метод:

```
Loc := ∅;
foreach (v ∈ V_G) do

  foreach (c ∈ Components_v)

    Loc[σ(c)] += E_v  ∩  {e | overline((e, typeof(e))) = c};
  done;
done;
```

Где $Components$ — корректное разбиение, $E_v \in E$ — клиентское рабочее множество (например, множество всех встречаемых в программе выражений). Последнее определяется нуждами использующего анализ оптимизирующего преобразования.

3.4 Параметризуемость алгоритма, сложность алгоритма

Внутрипроцедурный анализ параметризуем, в первую очередь, выбранным алгоритмом решения задачи нахождения инвариантного состояния. Также можно сказать, что он параметризуется видом функции $distance$. При реализации внутрипроцедурного алгоритма существует определенная свобода в степени точности алгоритма. Например, реализация функции $distance$ с учетом только информации, предоставляемой инвариантным состоянием, приводит к тому, что результаты решения задачи нахождения загрубления суперсостояния не информативнее результатов решения задачи поиска инвариантного состояния. В предложенном алгоритме используется другая, более точная функция $distance$. Для неё верен следующий факт.

Утверждение 1. Для программы π и её произвольного оператора v и разбиения на компоненты $Components$, построенного по формуле

$$\overline{(e_0, t_0)} = \{i | \exists (e,t) \in E \times T \ \exists I \ \alpha_I(e,t) = i, distance((e_0,t_0),(e,t)) \neq \perp\}$$

при условии транзитивности функции $distance$ верно, что построенное разбиение $Components$ корректно (определение корректности дано в Части 3.3.2).

Из транзитивности функции $distance$ следует, что полученное множество компонент действительно будет разбиением. То, что разбиение корректно, следует из независимости $distance$ от интерпретации I.

Рассмотрим сложность шага первой части предложенного алгоритма. Вычисление $super_v^\pi$ включает объединение информации по входящим дугам ($s\prime$) и вычисления для возможных значений правой части (обозначенное в Части 3.2 β). Таким образом, суммарная сложность выполняемых операций для каждого i есть

$$\underline{O}(S_i \cdot |in(v)| + S_j)$$

где S_i — объем задачи для данного i на предыдущих шагах. Откуда алгоритм для данного i экспоненциально зависит от длины пути в уграфе, т.е., в худшем случае, от числа вершин. i ограничено максимальной мощностью разбиения на компоненты, которое, в свою очередь, в худшем случае равно числу различных выражений в процедуре. Из преведенного выше, оценка времени работы и объема данных алгоритма для худшего случая есть $\underline{O}(2^{|V_G|} \cdot |E|)$ (в предположении хэшированного доступа к данным алгоритма), где V_G — число вершин уграфа, E — выражения, встречаемые в программе. Как было отмечено в обзоре, потоко-независимые анализы имеют сложность $\underline{O}(|E|)$. Однако, среднее и лучшее времена работы для предлагаемого алгоритма составляют $\underline{O}(|V_G| \cdot |E|)$ (при равномерном росте объема задачи на всех путях) и $\underline{O}(|E|)$ (на линейном участке со всей информацией, появляющейся в последнем операторе).

Сложность второго шага также достаточно просто оценить — это суммарный объём задачи нахождения суперсостояния по всем узлам графа, откуда он мажорируется сложностью построения решения этой задачи.

Таким образом, для небольших функций, а также функций, построенных, например в результате встраивания вызовов функций, когда задача наращивается отдельно для каждого случая встраивания (т.е., примерно выполняются условия среднего времени работы), теоретическая сложность алгоритма может считаться приемлимой.

3.5 Преодоление в рамках алгоритма стандартных трудностей

В Главе 2 были названы ряд объектов, которые вызывают трудности при реализации алгоритма анализа указателей. Для внутрипроцедурного анализа это:

- Рекурсивные структуры данных
- Динамическое выделение памяти

Последовательно разберём, какой подход был выбран в каждой из перечисленных ситуации.

Рекурсивные структуры данных. Информация о рекурсивных структурах данных сохраняется в контектсте и обрабатывается при создании энергичных (содержащих вычисленные значения) структур с информацией о синонимах. Выбор глубины анализа и методов построения остается за реализацией клиентской части алгоритма. Для применения анализа указателей в DCE [1] (оно описано в Части 4) реализация использовала k-ограничение с высотой, равной двум. Таким образом, для рекурсивных структур алгоритм не осуществляет загрублений, характерных для большинства других алгоритмов.

Динамическое выделение памяти. Выбранное представление информации о суперсостоянии для значений, создаваемых динамически, ставит в соответствие неопределённое значение $distance$. Это необходимо для обеспечения сходимости алгоритма построения инвариантного состояния. Это пессимизирует анализ при наличии активной работы с многократно явно выделямой памятью.

Итого, для внутрипроцедурного анализа для "трудных" объектов были предложены методы работы в рамках алгоритма, позволяющие получать для них содержательную информацию.

[1] DCE (Dead Code Elimination, удаление ненужного кода), данное оптимизирующее преобразование подробно описано в [1].

Глава 4

Реализация анализа указателей

Реализация приведенного выше алгоритма анализа указателей выполнена в виде набора функций и классов на языке C++ как часть оптимизатора для свободно распространяемого front-end'а компилятора **lcc**, оптимизатор носит название *Promise*.

Результатом работы front-end'а являются файлы, хранящие внутреннее представление, которое называется pickle[1]. Среда Promise считывает pickle файлы и строит на основе внутреннего представления **lcc** собственное. Над данным представлением производятся оптимизирующие преобразования. Promise позволяет осуществлять ряд внутри- и межпроцедурных преобразований. Выходом оптимизатора является pickle файл с оптимизированным модулем (программой).

С точки зрения реализации анализа указателей оптимизирующие преобразования выступают в роли *клиентов*, которые инициируют построение контекстов, описанных в Главе 3. В процессе построения контекстов выполняется непосредственно анализ указателей, результаты его работы сохраняются как часть информации, содержащейся в контекстах. Далее эта информация используется клиентами для произведения более эффективных преобразований.

Ниже более подробно изложены детали реализации предлагаемого алгоритма. Структура изложения следующая:

- Описание особенностей работы первой части алгоритма, т.е., решения загрубления задачи построения суперсостояния.

- Описание особенностей работы второй (клиентской) части алгоритма на примере её реализации для задачи поиска definition-use зависимостей и DCE.

[1] Подробно формат внутреннего представления и его сериализации в файл описан в технической документации, поставляемой с **lcc**.

4.1 Описание особенностей работы первой части алгоритма

Наиболее интересными с точки зрения реализации представляется работа с pickle представлением, которое имеет ряд особенностей по сравнению с традиционными представлениями, на которых работают большинство алгоритмов анализа указателей.

4.1.1 Особенности работы с внутренним представлением

Внутреннее представление **lcc** имеет ряд особенностей, которые не присущи представлениям, на которых работают перечисленные в Главе 2 алгоритмы, использующие более высокоуровневое представление, над которым не производится каких-либо предварительных преобразований.

Приведём основные из этих особенностей:

- Областью видимости переменной всегда является процедура, что избавляет от необходимости учета локальных областей видимости при внутрипроцедурном анализе. Например:

  ```
  int f() {
    int a, *p;
    a = 0;
    p = & a;
    {
      int a;
      p = & a;
    }
      *p = 1;
    return *p;
  }
  ```

  В данном случае при запоминании информации о синонимах в операторе return, нет необходимости вводить различие между `a` во внешнем блоке и `a` во вложенном блоке, усложняя работу со структурами данных, хранящими информацию о синонимах. Внутреннее представление **lcc** гарантирует различные численные идентификаторы для этих переменных. Кроме того, подобное представление позволяет считать информацию об адресах переменных потоко-независимой. Тем самым, например, информация о том, что `p` указывает на адрес второй переменной `a`, остается корректной в процессе анализа и по выходе из области видимости второго объявления `a`.

- Переменным присвоены уникальные для области видимости, создаваемой данной процедурой, численные идентификаторы (в том числе, всюду уникальные идентификаторы глобальных переменных). Это позволяет использовать их идентификаторы для нужд функций хэширования.

- Произведено частичное удаление недостижимого кода (проанализированы на явную константность условия структурных операторов и т.д.), например:

  ```
  int f() {
    int a, *p;
    a = 0;
    p = & a;
    if (1==0) {
      *p = 1;
    }
    return *p;
  }
  ```

  В данном случае условный оператор будет удален, что позволяет точно установить значение выражения `*p` в операторе `return *p`.

- Доступ к полям структур заменён операциями над адресными выражениями. Данное преобразование, производимое при создании внутреннего представления **lcc**, упрощает структуру программы. Например, программа:

  ```
  struct A {
    int a;
    struct A * p;
  };
  int main() {
    struct A a;
    a.p = & a;
    return 0;
  }
  ```

  была преобразована **lcc** в программу, которую можно записать на языке C следующим образом:

  ```
  int *l5, **l6;
  l6 = (int**) (&l5 + 1);
  *l6 = l5;
  ```

- Предоставление информации о размещении экземпляров составных типов в памяти. Как известно, стандарт ANSI языка C накладывает достаточно нежесткие условия на реализацию представления в памяти составных типов. Так, например, про структуру можно сказать только следующее: её размер будет не меньше суммарного размера её полей; адрес её первого поля будет совпадать с адресом структуры, соответственно, оно будет расположено в структуре первым; если две структуры имеют в начале последовательности полей, каждое из которых имеет тот же тип, что и его пара в другой последовательности, то эти поля будут размещены в памяти с одинаковыми смещениями от начал структур. Очевидно, что анализ, использующий только информацию, доступную из стандарта, меннее эффективен, чем анализ, использующий точные данные о смещениях и размерах полей в структуре.

- В тексте присутствуют ряд оптимизаций, проведенных front-end'ом. В частности, собраны общие подвыражения, построены статические синонимы для вырезок из массивов с константным смещением. Наличие общих подвыражений незначительно ускоряет операции работы с термами. Однако оно затрудняет повторный CSE[1] и усложняет реализацию структур данных ряда других преобразований. Таким образом, данная особенность внутреннего представления скорее является недостатком, чем преимуществом с точки зрения оптимизирующих преобразований, но не оказывает заметного влияния на работу анализа указателей.

Подводя итоги, можно сказать, что внутреннее представление **lcc** более удобно для алгоритма анализа указателей, чем традиционные представления, используемые большинством других алгоритмов.

4.2 Особенности реализации клиентской части алгоритма

Многие внутрипроцедурные оптимизирующие преобразования могут менять уграф процедуры, тем самым уточняя доступную информацию о синонимах. Используя возможность постепенного построения контекстов, клиенты могут повысить эффективность анализа указателей. Поэтому важен своевременный вызов анализа указателей, чтобы полученная информация была бы максимально актуальной. Также отдельные преобразования могут не использовать часть информации о синонимах, причем критерии отбора информации могут быть различными. Типичным примером может являться загрубление клиентом информации о циклических структурах, которое может быть различным и производиться, например, из эвристических соображений клиента. В связи с этим, выделена клиентская часть алгоритма, которая реализуется в соответствии с запросами преобразования. Ниже приведен пример реализации клиентской части для типичной внутрипроцедурной задачи, возникающей при применении оптимизирующих преобразований.

4.2.1 Встраивание в задачу нахождения definition-use зависимостей

Приведём пример встраивания анализа указателей в решение задачи нахождения definition-use зависимостей. Как известно, например, из [9], где приводится полная формализация и решение данной задачи, решение этой задачи сводится к прямой задаче анализа потоков данных, где в качестве

[1] CSE (Common Subexpressions Elimination) — удаление общих подвыражений, преобразование подробно описано в [1].

элементов полурешётки выступают множества REF и MOD — множества используемых и модифицирумых переменных в данном операторе. Рассмотрим, как меняются процедуры нахождения множеств REF и MOD при использовании данных анализа указателей.

Код процедур до применения анализа указателей выглядит следующим образом[1]:

```
function Ref (Operator v)
{
  res = {};

  switch (v->type)
  {
    case assign:
      {
        lval = P<assign>(v).left;
        rval = P<assign>(v).right;

        if ( RVal(lval).contains (INDIR) ||
rval.contains(INDIR) )
        {
          return ANY; // консервативная оценка
        }
      }

    case test :
    case stop :
      if (v->expressions.cointains (INDIR) )
      {
        return ANY; //консервативная оценка
      }
      foreach ( use, v->expressions )
      {
        res += ValueNumber (use);
      }
  }
  return res; // список потоко-независимых номеров
}

function Mod (Operator v)
{

  if (v->type == assign)
  { // только присваивание модифицирует переменные
    lval = P<assign>(v).left;

    if ( lval.contains (INDIR) )
```

[1] Из соображений экономии места приведён приведён упрощённый вариант, в котором опущены технические детали, связанные с представлением и обработкой операторов и выражений программы.

```
    {
      return ANY; // консервативная оценка
    }
    return ValueNumber (lval); // потоко-независимый номер
  }
}
```

В тексте процедуры после внесения изменений, учитывающих данные анализа указателей, консервативная оценка заменяется на более точную. Приведём примерный вид функций Mod и Ref с использованием упомянутой выше техники k-ограничения для обрывания обработки множественных разыменований.

```
function Ref (Operator v)
{
  res = {};

  switch (v->type)
  {
    case assign:
      {
        lval = P<assign>(v).left;
        rval = P<assign>(v).right;

        if ( RVal(lval).contains (INDIR(INDIR))
         || rval.contains(INDIR(INDIR)) )
        { // консервативная оценка для большей высоты
          return ANY;
        }
        if ( RVal(lval) == INDIR(val) )
        { // возвращаются возможные синонимы
          res += Aliases (lval);
        }

        if (rval.contains(INDIR) )
        {
          foreach ( use, v->expressions )
        { // возвращаются возможные синонимы
            res += Aliases (rval);
        }
      }

    case test :
    case stop :
      if (v->expressions.cointains (INDIR(INDIR)) )
      { //консервативная оценка для большей высоты
        return ANY;
      }
      foreach ( use, v->expressions )
      {
        res += Aliases (use);
```

```
    }
  }
  return res; // список потоко-независимых номеров
}

function Mod (Operator v)
{

  if (v->type == assign)
  { // только присваивание модифицирует переменные
    lval = P<assign>(v).left;

    if ( lval.contains (INDIR(INDIR)) )
    {
      return ANY; // консервативная оценка
    }
    return Aliases (lval); // потоко-независимый номер
  }
}
```

Функция Aliases, заменившая в приведенном коде функцию ValueNumber, производившую потоко-независимую нумерацию выражений, представляет собой реализацию клиентской части предлагаемого алгоритма, использующую отложенное вычисление синонимов. Результатом функции является список потоко-независимых номеров выражений. В текущей реализации в теле функции Aliases вызывается функция ValueNumber.

Рассмотрим, как изменяется функция перехода для данной задачи. Слияние информации по двум входящим дугам не меняется, прохождение через оператор изменяется только для оператора присваивания. Если без применения результатов анализа указателей процедура прохода через оператор присваивания выглядела следующим образом:

```
procedure Trans(assign v)
{
  if ( v.left != INDIR )
  {
    assigns[ValueNumber(v.left)] = UNREACHABLE;
  }
   // иначе консервативная оценка --
   // все операторы остаются достижимыми
}
```

то теперь есть возможность переписать её следующим образом:

```
procedure Trans(assign v)
{
  foreach (alias, mustAliases(v.left))
  {
    assigns[ValueNumber(alias)] = UNREACHABLE;
  }
}
```

где функция mustAliases возвращает обязательные синонимы выражения и может иметь следующую логику:

```
function mustAliases(Expression e)
{
  if ( aliases(e).contains(ANY) ) return {e}
   // один дополнительный к e may-alias - это must-alias:
  if ( aliases(e) <= 2 ) return aliases(e);
  return {e};
}
```

Если изначально реализовать второй вариант функции Trans, заменив mustAliases заглушкой, всегда возвращающей $\{e\}$, то встраивание в решение задачи поиска definition-use зависимостей становится достаточно простым и локализуется построением множеств REF и MOD, необходимых и в других задачах оптимизации (например, при экономии вызовов).

4.2.2 Встраивание в DCE

Встраивание в задачу нахождения definition-use зависимостей позволяет использовать данные анализа указателей в оптимизациях, для которых необходима данная информация. Одной из таких задач является DCE.

В простейшем варианте, DCE устраняет ненужные присваивания. При этом используется информация, получаемая при анализе definition-use зависимостей. Присваивания, не являющиеся определениями ни для одного использования, удаляются из программы. Данное преобразование не требует дополнительных действий при встраивании анализа указателей, так как использует его результаты опосредованно, через результаты построения definition-use зависимостей.

В Главе 5 приведён ряд практических результатов, полученных для такого вида DCE как на тестовых примерах, так и при оптимизации прикладных программ.

4.2.3 Встраивание в другие оптимизирующие преобразования

Встраивание в другие оптимизирующие преобразования также достаточно просто. Для *CSE* встаивание проводится аналогично встраиванию в DCE, т.е., фактически, дополнительных действий, кроме подмены функции вычисления синонимов данного выражения не требуется. *Экономия вызовов* — преобразование, сохраняющее значения вызовов функции без побочных эффектов с повторяющимися аргументами, использует множества MOD и REF для определения потенциальной возможности экономии той или иной функции. Встраивание в данное оптимизирующее преобразование также не требует дополнительных действий, кроме замены функции ValueNumber межпроцедурным аналогом. Встраивание в данные преобразования находятся в процессе реализации.

В целом, следует отметить простоту действий, необходимых для использования результатов реализации предлагаемого алгоритма в оптимизирующих преобразованиях.

Глава 5

Экспериментальные результаты

В данной главе приводится сравнительная информация о росте эффективности оптимизирующих преобразований, произошедшем в результате применения предлагаемого алгоритма анализа указателей.

5.1 Использование в DCE и встраивании вызовов функций

DCE — одна из классических оптимизаций, значительно выигрывающих при использовании анализа указателей для С-подобных языков. Задача преобразования заключается в удалении операторов программы, выполняющих действия, результаты которых не используются в последующих операторах. Ранее уже давалась ссылка на литературу, из которой можно узнать постановку задачи и алгоритм реализации данного преобразования. DCE в оптимизаторе **lcc** удаляет лишние присваивания (т.е., присваивания, выполнение которых не влияет на результаты выполнения программы).

Без применения анализа указателей степень грубости результатов DCE можно оценить на следующем примере кода:

```
0: int f() {
1:    int a, *p;
2:    a = 1;
3:    p = & a;
4:    a = 2;
5:    *p = 3;
6:    return a;
7: }
```

Присваивание 4 не может быть удалено из-за консервативного предположения о том, что p может указывать на произвольную область памяти, не обязательно на переменную a. Таким образом, данная процедура оптимизировалась следующим образом:

```
0: int f() {
1:    int a, *p;
```

```
3:    p = & a;
4:    a = 2;
5:    *p = 3;
6:    return a;
7: }
```

Применение анализа указателей позволяет в данном случае получить более короткий код:

```
0: int f() {
1:    int a, *p;
3:    p = & a;
5:    *p = 3;
6:    return a;
7: }
```

Экономятся два оператора присваивания, а не один, как в случае консервативных предположений о поведении указателей.

Ниже приводятся данные об эффекте применения анализа указателей в DCE, полученные при применении как на тестах, так и на прикладных программах.

Тесты составлялись путём выполнения встраивания кода (inline) всех функций в точки вызова. Данный выбор оправдан тем, что имитирует практические ситуации, возникающие при работе оптимизатора, а не искуственные ситуации, например, написание ручных заведомо неоптимальных тестов, при котором тот или иной алгоритм ведёт себя лучше или хуже. Практика также показывает, что к реальным программам без предварительных оптимизирующих преобразований удаление недостижимого кода неприменимо (в том смысле, что изменений в программе при этом не происходит).

Рассмотрим ситуацию встраивания кода в точки вызовов малых по объёму функций, принимающих параметры по указателю/ссылке. Ниже приведена таблица для простейших примеров. В первой колонке представлен исходный код, во второй — результат после встраивания кода процедур и распространения копирующих присваиваний, в третей — результат применения DCE с анализом указателей. Во всех данных примерах без анализа указателей DCE не изменяет программу.

Исходная программа	*Результат inline*	*Результат после DCE*

`int sqr(const`
`double & d)`
`{`
`return d * d;`
`}`
 	
`int main() {`
`int x;`
`x = 1;`
`return sqr(x);`
`}`
	`int main() {`
	`int l9, *l13,`
	`l15, l14,`
	`l10;`
	
	`l9 = 1;`
	`l13 = &l9;`
	`l14 = *l13;`
	`l15 = l14 * l14;`
	`l10 = l15;`
	`return l10;`
	`}`
		`int main() {`
		`int l9, *l13,`
		`l15, l14;`
		
		`l9 = 1;`
		`l13 = &l9;`
		`l14 = *l13;`
		`l15 = l14 * l14;`
		
		`return l15;`
		`}`

`int`
`mystrlen(char* s)`
`{`
`char* p;`
`for (p=s;`
`*p++;);`
`return p-s-1;`
`}`
 	
`int main() {`
`char* s =`
`"aliases";`
 	
`return`
`mystrlen(s);`
`}`
	`char* g11 =`
	`"aliases";`
	`int main() {`
	`char *l14, *l18,`
	`*l19, *l20;`
	`int l21, l15;`
	
	`l14 = g11;`
	`l18 = l14;`
	`l19 = l18;`
	`l20 = l19;`
	`l19 = *l20 + 1;`
	`while(*l20 != 0)`
	`{`
	`l21 = int(l19-`
	`l18)-1;`
	`}`
	`l15 = l21;`
	`return l15;`
	`}`
		`char* g11 =`
		`"aliases";`
		`int main() {`
		`char *l14, *l18,`
		`*l19, *l20;`
		`int l21, l15;`
		
		`l14 = g11;`
		
		`l19 = l14;`
		`l20 = l19;`
		`l19 = *l20 + 1;`
		`while(*l20 != 0)`
		`{`
		`l21 = int(l19-`
		`l18)-1;`
		`}`
		
		`return l21;`
		`}`

Из данной таблицы видна актуальность как проведения DCE после встраивания, так и необходимость обладания информацией об указателях для успешного DCE.

Были также проведены тесты для случая рекурсивных структур и динамического выделения памяти. Ниже приведен один из тестов на рекурсивные структуры данных. Как уже отмечалось, анализ циклических структур в рамках предложенного алгоритма может быть сколь угодно точным.

Исходная программа	*Результат inline*	*Результат после DCE*

<pre>struct A {
  int data;
  struct A* next;
}
void assign1(
      struct A* a)
{
  a->data = 1;
}
 	
int main()
{
  struct A a1, *p;
 	
  a1.next =&a1;
  p = &a1;
  assign1(p);
  p = p->next;
  return p->data;
}</pre>
	<pre>int main()
	{
	  int l11[2],
	      *l10, l12;
	
	  l12 = l11;
	  l10 = l11;
	  *l10 = 1;
	  l10 = (int*)
	      *(l10+1);
	
	  return *l10;
	}</pre>
		<pre>int main()
		{
		  int l11[2],
		      *l10, l12;
		
		  l12 = l11;
		  l10 = l11;
		  *l10 = 1;
		
		  return *l10;
		}</pre>

Последнее присваивание удалено в следствие того, что значение левой части не изменилось в результате присваивания. Данная информация характерна только для предлагаемого алгоритма, и не может быть получена при работе других алгоритмов анализа указателей.

В качестве прикладных программ были использованы популярные кодировщики-декодировщики информации. Это программа под условным названием gzip, известная GNU реализация архиватора zip, программа bzip, также GNU архиватор на основе zip'а, предоставляющий ряд дополнительных возможностей, и ranking, программа выполняющая сжатие файлов при помощи одноимённого алгоритма. Данные прикладные программы характеризуются большим объёмом целочисленных вычислений, первоначально качественным с точки зрения эффективности исполнения кодом каждой отдельной процедуры.

Результаты показали, что в прикладных программах до встраивания вызовов ненужный код отсутствует. Для анализа эффективности анализа указателей производились связывание отдельных pickle файлов в общий файл, выборочное встраивание вызовов функций, затем сравнивались результаты работы после обычного DCE и DCE, использующего информацию о синонимах, полученную при помощи предлагаемого алгоритма анализа указателей.

Ниже даны результаты тестирования некоторых модулей ranking. В таблице даны имена исходных модулей, указан размер первоначально полученных pickle файлов, количество произведённых встраиваний функций и суммарное количество удалённых присваиваний во всех функциях после выполнения DCE с применением анализа указателей (для сравнения в скобках даны результаты DCE без применения анализа указателей).

Имя модуля	Размер, байт	Количество inline	Удаленные присваивания
lists.s	7046	1	1(0)
ranking.s	6247	2	0(0)
reader.s	7461	6	2(0)
writer.s	7951	6	0(0)

К сожалению, осуществить достоверное тестирование увеличения быстродействия программ осуществить не удалось, поэтому временные характеристики не приводятся. Инспекция ассемблерного кода свидетельствует об экономии в результирующем коде в том числе и операций в многократно выполняющемся цикле, что, несомненно, должно уменьшать время работы программы на процессоре.

В целом, результаты применения предлагаемого алгоритма свидетельствуют о перспективности его применения для порождения оптимизированного кода. Однако, в оптимизаторе для компилятора общего назначения использование алгоритма проблематично из-за больших накладных расходов по времени при его использовании (замеры показывают, что анализ указателей занимает до половины и более времени работы оптимизатора). Алгоритм выгодно применять при необходимости получения оптимизированного кода в системах с ограниченной работой с динамической памятью (при таких условиях алгоритм выдаёт наиболее приближенные к точным результаты).

Глава 6

Заключение

Эффективное решение задачи анализа указателей позволяет улучшить качество преобразований программы. Без анализа указателей не обходятся современные компиляторы. Текущие промышленные решения ограничиваются внутрипроцедурным потоко-независимым анализом, использующим почти линейные алгоритмы. Данные алгоритмы недостаточно эффективны для оптмизирующих компиляторов, стремящихся максимально полно исползовать целевую архитектуру (компиляторы для мобильных платформ, платформ с параллелизмом на уровне инструкций и т.д.). Потокозависимый анализ представляет собой область, в которой активно ведутся исследования. Различными авторами предлагается ряд алгоритмов, которые могут считаться в большей или меньшей степени удачными. Существует также опыт сравнительных исследований алгоритмов анализа указателей.

Работа предлагает собственный вариант потоко-зависимого анализа, который выполнен в рамках настраиваемого окружения, может быть использован как в рамках некоторого межпроцедурного анализа, так и самостоятельно. Проведено теоретическое обоснование алгоритма, изучены вопросы теоретической сложности и эффективности алгорима.

Представлена реализация алгоритма как части оптимизатора, лицензию на распространение которого имеет фирма Image Craft (www.imagecraft.com), для свободно распространяемого front-end'а компилятора **lcc**. Результаты работы анализа указателей используются в данном оптимизаторе для нужд оптимизирующих преобразований программ. В том числе, для таких, как удаление ненужного кода, удаление общих подвыражений, экономия вызовов и т.д.

Проведен ряд тестов для изучения эффективности предлагаемого алгоритма. Среди тестов присутствуют тесты на рекурсивные структуры данных, динамически выделяемые данные и другие.

Указаны возможные направления дальнейших исследований на тему анализа указателей.

Литература

[1] Muchnick S. *Advanced Compiler Design & Implementation*. pp. 857, Morgan Kaufmann, San Fran., CA, 1997.

[2] *GNU C compiler reference*. Available at www.gcc.gnu.org.

[3] Horwitz S. *Precise flow-insensitive may-alias analysis is NP-hard*. ACM Transactions on Programming Languages and Systems Vol 19 No 1, Jan. 1997 .

[4] Landi W. *Undecidability of static analysis*. ACM Letters on Programming Languages and Systems Vol 1 No 4 (Dec. 1992), pp. 323–337 .

[5] Hind M., Pioli A. *Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses*. Available at ibm.com/people/h/hind/RC21251.ps.

[6] Andersen L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] Steensgaard B. *Points-to analysis in almost linear time*. Symposium on Principles of Programming Languages, pp. 32–41, 1996.

[8] Shapiro M., Horwitz S. *Fast and accurate flow-insensitive point-to analysis*. 24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, pp. 1–14, Jan. 1997. .

[9] Jones N., Muchnick S. *Program Flow Analysis: Theory and Applications*. pp. 131, Prentice Hall, 1979.

[10] Deutsch A. *Interprocedural may-alias analysis for pointers: Beyond k-limiting*. SIGPLAN Conference on Programming Languages Design and Implementation, pp. 230–241, New York, NY, 1994. ACM PressProgram Flow Analysis: Theory and Applications .

[11] Choi J.-D., Burke M., Carini P. *Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects* . 20th Annual ACM Symposium on Principles of Programming Languages, pp. 232–245, Jan. 1993 .

[12] Landi W., Ryder B. G. *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 235–248, 1992 .

[13] Hind M., Burke M., Carini P., Choi J. *Interprocedural pointer alias analysis.* ACM Trans. Programming Languages and Systems, 21(4) pp. 848–894, May 1999 .

[14] Wilson R. P., Lam M. S. *Efficient, Context-Sensitive Analysis for C Programs.* ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, pp. 1-12, Jun. 1995 .