Санкт-Петербургский Государственный Университет
Математико-Механический факультет

Кафедра системного программирования

Потоко-чувствительный анализ указателей, основанный на диаграммах двоичных решений

Дипломная работа студента 544 группы
Тимофеева Н.М.

Научный руководитель	.... /подпись/	к.ф.-м.н. Д.Ю. Булычев
Рецензент	.... /подпись/	Е.Н. Вигдорчик
"Допустить к защите" зав. кафедры	.... /подпись/	д.ф.-м.н. А.Н. Терехов

Санкт-Петербург
2007

Содержание

1 Введение **3**

2 Существующие методы анализа указателей **5**
2.1 Внутрипроцедурный анализ 6
2.2 Межпроцедурный анализ 8

3 Диаграммы двоичных решений **10**

4 Описание предлагаемого подхода **13**
4.1 Абстрактный язык 13
4.2 Абстрактная память 13
4.3 Условные значения 14
4.4 Семантика исполнения с условными значениями 14
4.5 Примеры 16
4.6 Сохрание условных значений в диаграмме двоичных решений 19

5 Алгоритм **22**

6 Реализация **23**

7 Заключение **26**

A Приложения **27**
A.1 Пример 27

1 Введение

Анализ указателей является одной из классических задач статического анализа. Его цель — построение оценки значений всех переменных, имеющих тип указатель. Для каждой переменной такого типа он вычисляет множество абстрактных ячеек памяти, на которые переменная может указывать во время исполнения программы. Это множество называется множеством синонимов.

Анализ указателей является неотъемлимой частью анализа потока данных для любого языка, их использующего. Например, анализ достигающих определений для выражения $*p = 13$ должен сделать пессимистичные предположения без наличия информации об указателе p: p может указывать на все доступные в данной области видимости переменные, которые в свою очередь должны быть удалены из результата. Анализ достигающих определений становится практически бесполезным. С другой стороны, если известно, что множество синонимов состоит из $\{(*p, x), (*p, y)\}$, то необходимо удалить определения только для переменных x и y.

Вне зависимости от того, как представляется множество отношений, его объем может расти как при уточнении, так и при загрублении решения, поэтому очень важным вопросом является поиск эффективного представления множества синонимов. В работе [WL04] показано, что для одного из видов анализа указателей требуется порядка 10^{24} байт. В настоящей работе для представления и работы с множествами отношений используется диаграмма двоичных решений (Binary Decision Diagram).

Существуют несколько групп алгоритмов решения задачи анализа указателей. Выделяют два уровня решения — межпроцедурный и внутрипроцедуный. Такое разделение связано с тем, что на уровне процедуры существует четкое представление программы с помощью графа потока управления (Control Flow Graph). Это позволяет проводить как потоко-зависимый, учитывающий информацию из CFG, так и потоко-независимый анализы. Для межпроцедурного анализа обычно используют граф вызовов. В зависимости от того учитывает ли анализ контекст вызова процедур или нет, различают контекстно-чувствительный и контекстно-нечувствительный анализы. Существующие подходы более подробно рассмотрены в разделе 2.

Заметим, что анализ указателей — довольно сложная задача. Точный потоко-независимый анализ в случае, когда допускается сколь угодно большой уровень косвенности адресных выражений, является NP-трудной задачей [Hor97]. Для потоко-зависимого анализа поиск точного решения является NP-трудным при отсутствии динамического выделе-

ния памяти, и становится неразрешимым при его наличии. Поэтому применяемые на практике алгоритмы строят некоторое загрубление точного анализа.

Данная работа предлагает решение внутрипроцедурной задачи потоко-чувствительного анализа указателей для языка C, реализованного в среде Creen. Особенностью алгоритма является представление отношений синонимичности с помощью отношений в диаграммах двоичных решений. С помощью отношений BDD моделируется дерево условных значений. В узлах хранятся специальные значения — предсказатели, а в листьях абстрактные адреса памяти, принимаемые указателем в зависимости от значений в узлах. Подробно это дерево рассматривается в разделе 4.3. Новый подход позволяет получить более точную информацию о переменных и следовательно значительно увеличить точность анализа в целом. Описание диаграммы двоичных решений, а также необходимых в дальнейшем операций расположено в разделе 3. В разделе 4.6 рассматривается описание алгоритма хранения отношений синонимичности, а также интерпретации операций с переменными в виде диаграмм двоичных решений. Раздел 5 содержит описание нового алгоритма анализа указателей.

2 Существующие методы анализа указателей

Традиционно, синонимы представляются как отношения эквивалентности над абстрактным размещением в памяти [AA86]. Например, множество синонимов для оператора $p = \&x$ представляется набором $\{(*p, x)\}$.

Алгоритмы анализа указателей, разработанные за последнее десятилетие, используют два основных способа представления множества синонимов — явное и компактное. Компактное представление позволяет сохранять множество синонимов более экономно, причем по нему с высокой точностью можно восстановить явное представление. Оно аналогично представлению PointsTo с ограничением по высоте 1.

Определение 1. *PointsTo граф* (PTG) — ориентированный граф, вершинами которого являются классы эквивалентности выражений программы, а дугой e соединяются те и только те классы, для которых верно, что существует представитель класса $beg\,(e)$, разыменование которого является возможным синонимом некоторого представителя класса $end\,(e)$, где $beg\,(e)$, $end\,(e)$ начало и конец дуги e. Отношение эквивалентности, по которому производится разбиение на классы, определяется произвольным образом в рамках нужд алгоритма, смысл класса — множество неразличимых для алгоритма обращений к памяти, абстрактная ячейка памяти.

Примером использования компактного представления, а также некоторой его модификации могут служить работы [And94] и [Ste96]. Явное представление применено в алгоритме [LR92]. В данной работе используется новый подход [Bry92], объединяющий точность явного и экономичность компактного представления.

Общая схема, по которой работает большинство алгоритмов:

Алгоритм 1. *Общая схема реализации анализа указателей*

```
Построить PCG
foreach procedure f in the PCG loop
  инициализировать множество синонимов f, доступной
  межпроцедурной информацией
endloop
repeat
  foreach procedure f in the PCG loop
    используя межпроцедурную информацию, вычислить
    внутрипроцедурную информацию;
```

```
    обновить межпроцедурные множества для процедур
    вызываемых или вызывающих f
  endloop

  обновить PCG используя новую информацию о синонимах
  foreach new procedure f added to the PCG loop
    инициализировать межпроцедурное множество синонимов f
  endloop
until сойдутся межпроцедурные множества и граф потока вызова
```

2.1 Внутрипроцедурный анализ

Потоко-чувствительный анализ Потоко-чувствительный анализ вычисляет множество синонимов применяя потоковую функцию к каждому узлу графа потока управления (CFG). Обычно эти вычисления производятся не на всем CFG, а на его подмножестве, называемом SEG (Sparse Evaluation Graph).

Определение 2. *SEG* (Sparse Evaluation Graph) — это четверка $\langle N_{seg}, E_{seg}, N_{entry}, N_{exit} \rangle$, где множество узлов N_{seg} состоит из:

- GenNodes, множество узлов CFG, где потенциально может измениться информация о синонимах;
- MeetNodes, множество узлов CFG, где информация о синонимах объединяется;
- N_{entry} и N_{exit}, две дополнительные вершины, такие что все остальные вершины лежат на пути от N_{entry} к N_{exit}.

E_{seg} множество дуг, соединяющих узлы из N_{seg} и представляющие поток управления в *SEG*

В работе [HP98] удалось добиться уменьшения мощности множества синонимов на 74%, при этом время анализа уменьшилось в 2.4 раза.

Для каждого узла строятся множества фактов синонимичности IN и OUT. В начале работы входное множество начальной вершины инициализируется информацией, известной на входе в процедуру. Для остальных узлов множества остаются пустыми. Функция перехода для вершины выглядит следующим образом:

$$IN(e) = \bigcup_{i \in Pred(e)} OUT(i)$$

$$OUT(i) = \begin{cases} IN(i), & i \text{ — вершина объединения} \quad (1) \\ f_i(IN(i)), & f \text{ — потоковая функция} \quad (2) \end{cases}$$

Обычно в функции перехода выделяют уничтожающую часть (KILL) и генерирующую часть (GEN). Наиболее интересным моментом является генерация множества KILL для оператора разыменования указателя. Результат этой оперции может привести к возникновению бесконечных цепочек. Одним из первых решение этой проблемы предложил Ахо [AA86]. В его работе уровень косвенности был ограничен единицей и рассматривались только переменные, размещенные на стеке. Этот метод был расширен для указателей в динамически выделяемой памяти в работе [CWZ90].

В работе Дойча [Deu94] была предложена полурешетка, параметризуемая численной решеткой с оператором для обрыва бесконечных цепочек. Отношения синонимичности предсталялись как пара пути доступа и целого числа, представляющего количество итераций. Удалось добиться обработки структур практически произвольной вложенности, за счет сохранения позиционного свойства структур в отношениях синонимичности. Например, отношение „i-ый элемент списка X является синонимом $(2i+1)$-ого элемента списка Y“ может быть выражено с помощью этой полурешетки.

В работе [HP98] для обрыва бесконечных цепочек предложена реализация анализа указателей на основе численной полурешетки Дойча. Символьные пути доступа, используемые в данном алгоритме, нормализуются путем создания временных переменных и свертки последовательно выполняющихся операций в итеративные конструкции с ограничениями на количество итераций. Отношение синонимичности задается на нормализованных путях и элементах численной решетки, представляющих количество применений отдельных операций в них.

Потоко-нечувствительный анализ Потоко-нечувствительный анализ не учитывает порядок и количество исполнений операторов процедуры, влияющих на значения указателей или другим способом меняющих информацию о синонимах.

Впервые алгоритм потоко-нечувствительного вычисления синонимов для языка LISP был предложен в работе [Lar89].

В статье Андерсена [And94] рассмотрены полиномиальные алгоритмы потоко-нечувствительного контекстно-зависимого и контекстно-независимого анализов языка C. В ходе работы алгоритм обходит все

операторы процедуры, которые используют адресные выражения, и добавляет в граф одну или несколько дуг, индуцируемых данным оператором. Алгоритм параметризуется максимальной степенью вершины PointsTo графа. Особенность алгоритма заключается в том, что каждое адресное выражение порождает собственный класс эквивалентности.

Штеенсгаард представил представлен почти линейный потоконезависимый алгоритм. Этот алгоритм дает больший объем множеств потенциальных значений для указателей, чем алгоритм Андерсена, т.к. максимальная степень вершины PointsTo графа ограничена единицей и на каждом шаге в граф добавляется только одна дуга.

Хорвиц и Шапиро [SH97] разработали два параметризуемых алгоритма, точность которых в худшем случае может варьироваться между алгоритмами Андерсена и Штеенсгаарда в зависимости от параметров. Второй алгоритм вызывает первый с различными параметрами в зависимости от результатов анализа.

Для того, чтобы аппроксимировать данные программы конечным множеством с объемом, мажорируемым размером входных данных алгоритма, используется два основных метода. Во-первых, существует метод k-ограничения, когда все адресные выражения, содержащие более k операций, считаются синонимичными. Во-вторых, может использоваться ограничение количества возникающих при анализе указателей классов эквивалентности для адресных выражений путем склеивания значений, создаваемых в программе динамически.

2.2 Межпроцедурный анализ

Аналогично тому как во внутрипроцедурном анализе используется граф потока управления, в межпроцедурном анализе часто используют граф вызовов (PCG)

Определение 3. *Граф вызовов процедур* (Procedure Call Graph) — граф, вершинами которого являются функции, а дуги соответствуют вызовам этих функций.

В работе [LR92] была предложена идея объединения графа потока управления и графа вызова процедур в единый граф управления для всей программы (Interprocedural Control Flow Graph).

Определение 4. *Контекст вызова процедуры* — состояние стека вызова процедур на момент ее исполнения.

На точность межпроцедурного анализа сильное влияние оказывает учет контекста вызова процедур. По этому признаку различают контекстно-зависимый и контекстно-независимый анализы.

Алгоритм контекстно-независимого анализа не используют информацию о контексте и просто объединяют результаты из различных точек вызова.

Существует насколько способов учета контекста вызова процедур. Основная проблема при этом — возможность образования бесконечного стека вызовов в случае рекурсии. Одним из алгоритмов борьбы с бесконечными цепочками, является метод k-ограничения (k-CFA). Идея алгоритма заключается в запоминании последних k контекстов [Shi91].

Альтернативой метода k-CFA является метод клонирования контекста. Алгоритм преобразует PCG в Invocation Graph. В нем каждому нерекурсивному вызову соответствует своя копия процедуры [WL04].

Другая важная проблема — передача параметров, а также побочные эффекты вызова процедуры. Для решения этой задачи выделяют специальные функции $ForwardBind$ и $BackwardBind$, преобразующие, соответственно, информацию из входного множества оператора вызова во входное множество процедуры и информацию выходного множества процедуры в данные для выходного множества оператора:

$$Entry\left(f\right) = \bigcup_{c=(g,f)\in PCG} ForwardBind_g^c\left(In_c\right)$$

$$Out\left(c\right) = BackwardBind_g^c\left(Exit_g\right); \ \ \forall c = (g,f) \in PCG$$

где g - вызывающая процедура, f - вызываемая процедура, а c - оператор вызова.

Некоторые анализы дополняют этот набор функций, специальными множествами, позволяющими уточнить работу межпроцедурного анализа информацией о потоке управления. С помощью таких множеств удается изменять PCG в соответствии с изменяющейся информацией о синонимах.

3 Диаграммы двоичных решений

Диаграмма двоичных решений — это представление логической функции с n-переменными. Ее можно рассматривать как множество бинарных векторов длины n. Множество включает ровно те вектора, для которых значение функции равно 1.

Физически диаграмма двоичных решений — это направленный ациклический граф (DAG). Он имеет две терминальные вершины 1 и 0 без сыновей. У каждой нетерминальной вершины два сына, называемые 0-наследник и 1-наследник. Для того, чтобы определить, содержится ли данный бинарный вектор в диаграмме двоичных решений, необходимо обойти DAG начиная с корня и проходить по 0-, 1-наследникам, в зависимости от значения каждого бита вектора. Если обход заканчивается в узле 1, то вектор принадлежит множеству, иначе не принадлежит.

Для наглядности рассмотрим классическое компактное представление множества синонимов, выраженное через диаграмму двоичных решений. Возьмем простую программу:

```
x = &y;
q = &x;
p = q;
```

Для нее компактное представление множества синонимов будет иметь вид:

$$\{(*p, x)\,; (*q, x)\,; (*x, y)\} \tag{3}$$

Такое множество в диаграмме двоичных решений может быть представлено с помощью отношений. Отношения задаются на двух доменах, домене переменных и домене абстрактных ячеек памяти. Для простоты будем считать, что память для переменных x, y, p, q выделяется статически. Используя 00 для кодирования $*p$ и x, 01 для $*q$, 10 для $*x$ и y множество 3 представится в виде множества отношений:

$$PointsTo \;=\; \left\{\underbrace{00}_{V}\underbrace{00}_{H};\ \underbrace{01}_{V}\underbrace{00}_{H};\ \underbrace{10}_{V}\underbrace{10}_{H}\right\}$$

Диаграмма двоичных решений, соответствующая этому множеству преведена на рис 3(a). Каждая группа битов представляет некоторый элемент множества. Непрерывной дугой изображены пути в 1-наследников, а пунктирной в 0-нследников.

Определение 5. Физическими доменами (доменами) будем называть группы битов, представляющие все элементы некоторого множества.

Например, в приведенном коде, первая группа из двух битов представляет домен переменных, а вторая домен абстрактного представления в памяти.

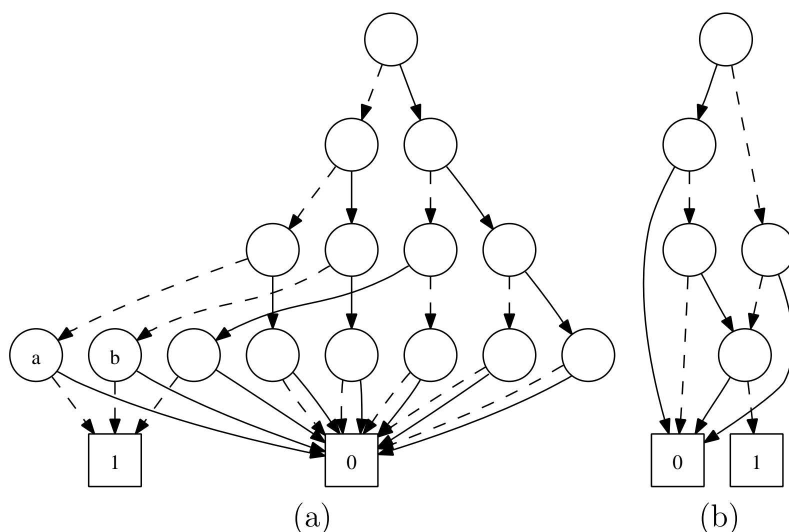

Рис. 1: (a) Диаграмма двоичных решений для компактного представления множества синонимов. (b) Сокращенная диаграмма двоичных решений для компактного представления множества синонимов

Можно заметить, что узлы, помеченные метками a, b, находятся на одном битовом уровне и у них совпадают сыновья, потому что они представляют одно и то же подмножество ячеек памяти. Их можно склеить в один узел. У некоторых узлов совпадают 0-наследник и 1-наследник; это означает, что значение бита не влияет на результат и его не нужно проверять, поэтому его можно удалить из дерева. В результате, если повторять описанные операции до тех пор пока не останется ненужных узлов или узлов, которые можно склеить, то получим сокращенную диаграмму двоичных решений 3(b).

В отличие от примера, в используемой в работе реализации диаграммы двоичных решений [Д.А06] сокращенное представление подреживается в ходе работы с ней.

Операции над диаграммами двоичных решений Для вычисления множества синонимов на диаграммах двоичных решений, помимо стандартных операций на множествах (объединение, пересечение, вычитание, дополнение) потребуются две дополнительные.

Операция *приписывания квантора существования* (existential quantification) убирает в диаграмме двоичных решений f зависимость от

некоторой битовой позиции b, путем конструирования функции, которая принимает значение $true$ только тогда, когда существует значение b, такое что значение f $true$. Например, если провести эту операцию для рассмотренной в этом разделе BDD по домену переменных, то получим:

$$exist\ PointsTo\ V \ \equiv\ \exists\ V\ PointsTo = \left\{ \underbrace{00}_{H}; \underbrace{00}_{H}; \underbrace{10}_{H} \right\}$$

Операция *переименования доменов* меняет один домен в отношении на другой. Эта операция очень полезна при интерпретации в диаграмму двоичных решений операций над указателями. Например, для того же множества:

$$rename\ PointsTo\ V\ V' \ =\ \left\{ \underbrace{00}_{V'}\,00; \underbrace{01}_{V'}\,00; \underbrace{10}_{V'}\,10 \right\}$$

4 Описание предлагаемого подхода

Перед тем как начать рассмотрение предлагаемого подхода, необходимо определить используемый абстрактный язык.

4.1 Абстрактный язык

Выделим следующие синтаксические группы:

$x \in Var$	множество переменных
$n \in N$	множество целых чисел
$a \in AExp$	арифметические выражения
$b \in BCond$	логические условия
$S \in Stmt$	операторы

Синтаксис рассматриваемого языка может быть описан следующим образом:

$$\begin{array}{rcl} a & \to & x \mid n \mid \&x \mid \; *a \mid malloc \\ S & \to & x \;:=\; a \mid skip \mid S_1;\; S_2 \mid \\ & & \text{if } (b)\; \{S_1\} \text{ else } \{S_2\} \mid \\ & & \text{while } (b)\; \{S\} \end{array}$$

В рамках данной работы логические условия — это некоторые абстрактные символы, которые интерпретируются специальным образом, изложенным ниже.

4.2 Абстрактная память

В настоящей дипломной работе память рассматривается как не более чем счетное множество

$$Mem = \{\bot\} \cup \{adr(x) | x \in Var\} \cup \ldots$$

Неформально говоря, $\bot$ соответствует неопределенному адресу, $adr(x)$ для переменной x соответствует адресу памяти, отведенной для данной переменной (все переменные считаются изолированными). Кроме того, существует потенциально бесконечное количество анонимных ячеек памяти.

4.3 Условные значения

Операторы в программе могут находиться внутри условных предложений и менять множество синонимов в зависимости от истинности некоторых условий. Для более точного вычисления множества значений переменных необходимо сохранять условия, при которых эти значения вычисляются.

Для представления множества возможных значений переменных было решено использовать технику, предложенную в диссертации А.Н.Терехова [А.Н76]. Именно, значение переменной представляется в виде дерева, промежуточные узлы которого хранят некоторые условия, ветви соответствуют значениями $true$ или $false$ для условия в вышележащем узле, а в листьях "собираются" значения переменных.

Для формального описания множества условных значений нам потребуется не более чем счетное множество элементарных условий

$$Cond = \{TRUE\} \cup \dots$$

которое обладает выделенным элементом $TRUE$, соответствующем начальному условию для программы. Множество $\overline{Cond}$ условий получается из $Cond$ по следующему правилу:

- $c \in Cond : c \in \overline{Cond}$;
- $c \in \overline{Cond} : !c \in \overline{Cond}$;
- $c, d \in \overline{Cond} : c\&d \in \overline{Cond}$;

Тогда множество условных значений $CVal$ описывается следующим образом:

- $n \in N : n \in CVal$
- $m \in Mem : m \in CVal$
- if (b) $\{val\}$, где $b \in \overline{Cond}$, $val \in CVal$

4.4 Семантика исполнения с условными значениями

Для описания семантики программы с условными значениями введем понятие состояния памяти как всюду определенной функции

$$S : Mem \to 2^{CVal}$$

причем $S(\bot) = \{\bot\}$. Можно определить значение условного выражения в данном состоянии следующим образом:

$$\overline{S}(m) = S(m), m \in Mem$$
$$\overline{S}(n) = \{\bot\}, m \in N$$
$$\overline{S}(if(C)\{val\}) = if(C)\{\overline{S}(val)\}, val \in CVal$$

Контекстом исполнения назовем пару (S, C), где S — состояние, $C \in \overline{Cond}$ — некоторое *текущее условие*, значение которого станет ясно чуть позже.

Опишем сематнику $A[a](S, C)$ арифметического выражения a в контексте (S, C):

- $A[x](S, C) = \{if(C)\{y\}|y \in S(adr(x))\}$;
- $A[n](S, C) = \{if(C)\{S(n)\}\}$;
- $A[\&x](S, C) = \{if(C)\{adr(x)\}\}$;
- $A[*a](S, C) = \{if(C)\{z\}|z \in \overline{S}(y), y \in A[a](S, C)\}$;
- $A[malloc](S, C) = \{if(C)\{new_S\}\}$;

Здесь через new_S обозначен некоторый элемент множества Mem, который не упоминается ни в каких частях условных значений области определения состояния S.

Неформально говоря, $A[a](S, C)$ есть множество условных значений выражения a в контексте (S, C).

Теперь опишем семантику исполнения операторов в данном контексте:

$SKIP$ $$\langle (S, C), skip\rangle \to S$$

$ASSIGN$ $$\langle (S, C), x := a\rangle \to S[mem(x) \leftarrow A[a](S, C)]$$

SEQ $$\frac{\langle (S, C), S_1\rangle \to S', \langle (S', C), S_2\rangle \to S''}{\langle (S, C), S_1; S_2\rangle \to S''}$$

$COND$ $$\frac{\langle (S, C\&C_b), S_1\rangle \to S', \langle (S, C\&!C_b), S_2\rangle \to S''}{\langle (S, C), if(b)\{S_1\}else\{S_2\}\rangle \to S' \sqcup S''}, \text{ где } C_b = new_C$$

$WHILE$
$\langle (S, C), while(b)\{S_1\} \rangle \rightarrow fix(\lambda(S, C) \rightarrow S' \sqcup S, \langle (S, C \& new_C), S_1 \rangle \rightarrow S')$

В двух последних правилах через new_C обозначено некоторое новое элементарное условие, которое не встречается ни в какой части условия C. Используемая там же операция объединения состояний определяется следующим образом:

$$\forall x \ (S_1 \sqcup S_2)(x) = S_1(x) \cup S_2(x)$$

Наконец, начальным контекстом исполнения программы считается контекст $(S_\perp, TRUE)$, где $S_\perp \equiv \perp$.

Очевидно, что множество контекстов является полурешеткой относительно операции $\sqcup$. В общем случае эта полурешетка имеет неограниченную высоту, что делает невозможным вычисление неподвижной точки, требуемое для определения множества условных значений произвольной программы. Для преодоления этой проблемы в данной работе использован метод k-ограничения. Именно, при попытке создания условного значения $if(c)\{val\}$, где высота val есть k, возвращается значение вида $val[x \leftarrow \perp]$, где x — самое "внутреннее" значение val.

4.5 Примеры

Рассмотрим пример:

```
ep := 3;
p := *ep;
q = malloc(3);
```

Начальное состояние S_0 описывается:

$$S_0(adr(ep)) \rightarrow \{if(true)\{\perp\}\}$$

$$S_0(adr(p)) \rightarrow \{if(true)\{\perp\}\}$$

$$S_0(adr(q)) \rightarrow \{if(true)\{\perp\}\}$$

Посчитаем $A[3](S_0, TRUE)$:

$$A[3](S_0, TRUE) = \{if(true)\{\perp\}\}$$

Получаем состояние S_1:

$$S_1(adr(ep)) \rightarrow \{if(true)\{\perp\}\}$$

$$S_1(adr(p)) \to \{if(true)\{\bot\}\}$$

$$S_1(adr(q)) \to \{if(true)\{\bot\}\}$$

Вычисляем $A[*ep](S_1, TRUE)$:

$$A[*ep](S_1, TRUE) = \{if(true)\{z\} | z \in \overline{S}y = \bot\} = \{if(true)\{\bot\}\}$$

Получаем состояние S_2:

$$S_2(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_2(adr(p)) \to \{if(true)\{\bot\}\}$$

$$S_2(adr(q)) \to \{if(true)\{\bot\}\}$$

Вычисляем правую часть для последнего оператора:

$$A[malloc(3)](S_2, TRUE) = \{if(true)\{mal_1\}\}$$

Получаем состояние S_3:

$$S_3(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_3(adr(p)) \to \{if(true)\{\bot\}\}$$

$$S_3(adr(q)) \to \{if(true)\{mal_1\}\}$$

$$S_3(mal_1) \to \{if(true)\{\bot\}\}$$

Рассмотрим пример с k-ограничением, где k = 2:

```
ep := 3;
p := &ep;
while (B)
{
 q = malloc(3);
}
```

Начальное состояние S_0 описывается, аналогично с прошлым примером:

$$S_0(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_0(adr(p)) \to \{if(true)\{\bot\}\}$$

$$S_0(adr(q)) \to \{if(true)\{\bot\}\}$$

Посчитаем $A[3](S_0, TRUE)$:

$$A[3](S_0, TRUE) = \{if(true)\{\bot\}\}$$

Получаем состояние S_1:

$$S_1(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_1(adr(p)) \to \{if(true)\{\bot\}\}$$

$$S_1(adr(q)) \to \{if(true)\{\bot\}\}$$

Посчитаем $A[\&ep](S_1, TRUE)$:

$$A[\&ep](S_1, TRUE) = \{if(true)\{adr(ep)\}\}$$

Состояние S_2:

$$S_2(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_2(adr(p)) \to \{if(true)\{adr(ep)\}\}$$

$$S_2(adr(q)) \to \{if(true)\{\bot\}\}$$

Начинаем итерироваться по циклу, первый раз считаем $malloc(3)$ в контексте $(S_2, true\&(q == \&p))$:

$$A[malloc(3)](S_2, true\&(B)) = \{if(true)\{if(B)\{mal_1\}\}$$

Новое состояние S_3:

$$S_3(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_3(adr(p)) \to \{if(true)\{adr(ep)\}\}$$

$$S_3(adr(q)) \to \{if(true)\{if(B)\{mal_1\}\};\ if(true)\{\bot\}\}$$

$$S_3(mal_1) \to \{if(true)\{\bot\}\}$$

Текущая высота оказалась равна k. Вычисляем $malloc(3)$ в контексте $(S_3, true\&(B)\&(B'))$:

$$A[malloc(3)](S_3, \ldots) = if(true)\{if(B)\{if(B')\{\bot\}\}\}$$

В итоге состояние S_4:

$$S_4(adr(ep)) \to \{if(true)\{\bot\}\}$$

$$S_4(adr(p)) \to \{if(true)\{adr(ep)\}\}$$

$$S_4(mal_1) \to \{\bot\}$$

$$\begin{aligned} S_4(adr(q)) \quad \to \quad & \{if(true)\{if(B)\{if(B')\{\bot\}\}\}; \\ & if(true)\{if(B)\{mal_1\}\}; \\ & if(true)\{\bot\}\} \end{aligned}$$

4.6 Сохрание условных значений в диаграмме двоичных решений

Хранение значений переменных в бинарном дереве ресурсоемко и поэтому в настоящей работе оно кодируется в сокращенную диаграмму двоичных решений.

Одно отношение синонимичности можно представить тройкой — имя переменной, логическое условие на значение памяти и абстрактное расположение в памяти. Для представления такого отношения в диаграмме двоичных решений потребуется 3 физических домена. Обозначим их за V, L, H соответственно.

Рассмотрим пример:

```
x = &y;
if (*x == 5) {q = &x} else {q = 0};
p = q;
```

Используя 00 для кодирования $*p$ и x, 01 для $*q$, 10 для $*x$ и y, 01 для 0. Логические условия сохраняем слудующим образом: $true$ — 00, $(*x == 5)$ — 01, $!\,(*x == 5)$ — 10. Множество синонимов, закодированное в диаграмму двоичных решений, выглядит следующим образом:

$$\{100010; 010100; 011001; 000100; 001001\}$$

В соответствии с выбранной кодировкой переменных, условий и памяти, логическая функция, представляющая эти элементы, будет иметь следующий вид:

элемент	код	логическая формула
x	10	$V_1 = 1\&V_0 = 0$
p	00	$V_1 = 0\&V_0 = 0$
q	01	$V_1 = 0\&V_0 = 1$
true	00	$L_1 = 0\&L_0 = 0$
$(*x == 5)$	01	$L_1 = 0\&L_0 = 1$
$!\,(*x == 5)$	10	$L_1 = 1\&L_0 = 0$
$\&x$	00	$H_1 = 0\&H_0 = 0$
$\&y$	10	$H_1 = 1\&H_0 = 0$
0	01	$H_1 = 0\&H_0 = 1$

Таблица 1: Кодировка элементов в терминах физических доменов

Points-to тройка выражается через конъюнкцию элементов доменов. Например, $x = \&y$ представляется формулой $V_1 = 0\&V_0 = 0\&L_1 =$

$0\&L_0 = 0\&H_1 = 0\&H_0 = 0$. Все множество отношений выражается через дизъюнкцию этих формул

$$PointsTo =$$

$$(V_1 = 1\&V_0 = 0\&L_1 = 0\&L_0 = 0\&H_1 = 1\&H_0 = 0)\,|$$
$$(V_1 = 0\&V_0 = 1\&L_1 = 0\&L_0 = 1\&H_1 = 0\&H_0 = 0)\,|$$
$$(V_1 = 0\&V_0 = 1\&L_1 = 1\&L_0 = 0\&H_1 = 0\&H_0 = 1)\,|$$
$$(V_1 = 0\&V_0 = 0\&L_1 = 0\&L_0 = 1\&H_1 = 0\&H_0 = 0)\,|$$
$$(V_1 = 0\&V_0 = 0\&L_1 = 1\&L_0 = 0\&H_1 = 0\&H_0 = 1)$$

Биты в диаграмме двоичных решений могут быть проверенны в том же порядке, в котором кодировались, $V_1V_0L_1L_0H_1H_0$. Однако используемый порядок может быть произвольным, единственное требование заключается в согласованности проверки битов всех элементов домена.

При использовании диаграмм двоичных решений очень важным вопросом является нахождение оптимального порядка. К сожалению, эта задача NP-трудна.

Помимо представления структур данных, операторы программы тоже могут быть представлены в терминах диаграмм двоичных решений. Рассмотрим операцию $x := p$, где x и p переменные. Для представления p используем старый домен V, а для переменной x и логического условия введем новые физические домены той же размерности V' и L' соответственно. Тогда

$$x := p \Leftrightarrow$$
$$AssignGen = (V_1 = 0\&V_0 = 0\&V_1' = 1\&V_0' = 0\&L_1' = 0\&L_0' = 0)$$
$$AssignKill = (V_1 = 1\&V_0 = 0\&L_1 = 0\&L_0 = 0)$$

Так как физический домен V общий для обеих диаграмм AssignGen и PointsTo, конъюнкция найдет в двух формулах все отношения, совпадающие на домене V

$$Gen = PointsTo\&Gen = \tag{4}$$

$$(V_1 = 0\&V_0 = 0\&V_1' = 1\&V_0' = 0\&L_1 = 0\&L_0 = 0\&L_1' = 0\&L_0' = 0\&H_1 = 0\&H_0 = 0)\,|$$
$$(V_1 = 0\&V_0 = 0\&V_1' = 1\&V_0' = 0\&L_1 = 1\&L_0 = 0\&L_1' = 0\&L_0' = 0\&H_1 = 0\&H_0 = 1)$$

Диаграмма двоичных решений для множества KILL вычисляется аналогично:

$$Kill = PointsTo\&Kill$$

Для вычисления нового множества PointsTo необходимо в (4) убрать зависимости от доменов V и L. Достаточно провести операцию “навешивания” кванторов существования по этим доменам:

$$\exists V \exists L PointsTo =$$

$$(V_1' = 1 \& V_0' = 0 \& L_1' = 0 \& L_0' = 0 \& H_1 = 0 \& H_0 = 0)\,|$$
$$(V_1' = 1 \& V_0' = 0 \& L_1' = 0 \& L_0' = 0 \& H_1 = 0 \& H_0 = 1)$$

Полученные формулы для GEN и KILL кодируют действие оператора присваивания на исходное множество PointsTo. Единственная проблема заключается в том, что в исходном множестве использовались домены V, L, H, тогда как в GEN V', L', H. Перед тем как его можно будет использовать необходимо заменить домены:

$$ReplaceGen = rename\,(\exists V \exists L PointsTo)\,V' \to V L' \to L =$$

$$(V_1 = 1 \& V_0 = 0 \& L_1 = 0 \& L_0 = 0 \& H_1 = 0 \& H_0 = 0)\,|$$
$$(V_1 = 1 \& V_0 = 0 \& L_1 = 0 \& L_0 = 0 \& H_1 = 0 \& H_0 = 1)$$

Окончательно:

$$PropagatedPointsTo = PointsTo \backslash Kill | ReplacedGen$$

5 Алгоритм

Описав операции диаграмм двоичных решений теперь можно приступить к изложению алгоритма анализа указателей. Анализ потоко-чувствительный, контекстно-независимый, внутрипроцедурный.

Для вычисления множества синонимов нам понадобится пять физических доменов, V, V', L, L' и H. Два дополнительных домена потребуются для представления отношений связи полей и структур.

Введем следующие диаграммы двоичных решений:

- $pointsTo \subseteq V \times L \times H$, множество отношений $pointsTo$;
- $mem \subseteq V \times H$, множество отношений связи переменных и абстрактной памяти.

Разберемся, как интерпретировать аппроксимацию для выражения e в терминах диаграмм двоичных решений.

Рассмотрим операцию разыменовывания указателя, а остальные будут реализовываться аналогично. Пусть выражение e имеет вид $*v$, где v — имя переменной. Закодируем v и контекстное условие в BDD и обозначим получившееся выражение за e_{bdd}. Память, на которую указывает v, вычисляется следующим запросом:

$$mem_{value} \;=\; exist\,(pointsTo\ \&\ e_{bdd})\ V$$

Используя BDD mem находим переменную с вычисленным адресом:

$$v_{mem} \;=\; mem\&mem_{value}$$

Наконец, остается только избавиться от домена H и получившийся результат можно будет использовать в дальнейших вычислениях:

$$v_e \;=\; (exist\ v_{mem}\ H)\ \subseteq V \times L$$

Благодаря удачно выбранной абстракции указателей, вычисление множества $pointsTo$ сводится к обработке операции присваивания, приведенной в разделе (4.6). Аргументы для нее интерпретируются в диаграмму двоичных решений аналогично описанной операции разыменовывания указателя.

6 Реализация

Одной из целей дипломного проекта было внедрение нового алгоритма в инструмент статического анализа *Creen*.

Краткое описание инструмента Creen *Creen* является инструментом статического анализа и рефакторинга программ, написанных на языке *C*. Он позволяет получить наглядное изображение структуры программы, удалить мертвый код, провести рефакторинг.

Основу инструмента составляет библиотека CIL [GCNW02](*C* Intermediate Language). Она используется для внтуреннего представления дерева разбора программы, написанной на языке *C*. Выбор этой библиотеки был сделан не случайно. Данная библиотека производит преобразования над программой, позволяя избавиться от множества сложных для анализов ситуаций. Например, оператор `i = i++ + 1;` раскрывается в список `i = i + 1; i = i++;`, делая граф потока управления более простым для анализа.

Реализация алгоритма анализа указателей Рассмотренный в данной дипломной работе алгоритм был реализован на языке OCaml, в виде набора параметризуемых модулей. Решение проектировалось таким образом, чтобы функтор, реализующий логику работы алгоритма, был языконезависимым.

Благодаря гибким возможностям функторов языка Ocaml была переиспользована библиотека двоичных решений из дипломной работы [Д.А06]. Решение абстрагировано от конкретной реализиции библиотеки двоичных решений и для перехода на новую достаточно изменить одну строчки кода:

```
module Relation = Relation.Make(Robdd)
```

Эффективность использования диаграмм двоичных решений сильно зависит от порядка проверки битов. В настоящей работе не исследовался вопрос выбора оптимальной нумерации и поэтому была реализована нумерация с прямым порядком проверки битов. Для ее изменения достаточно реализовать модуль со следующим интерфейсом:

```
module Numerator =
  sig

  (* Определяет тип элемента, кодируемого в
     диаграмму двоичных решений *)
```

```
  type t;

  (* Получает уникальный номер для элемента *)
  val entity2num : t -> int
  (* Восстанавливает элемент по номеру *)
  val num2entity : int -> t

  (* Для данного элемента строит набор битов *)
  val entity2boolarr : t -> bool array
  (* Восстанавливает элемент по данному набору битов *)
  val boolarr2entity : bool aray -> t

  val toString : t -> string

  end
```

Необходимо отметить, что нумерация для переменных, предсказателей и памяти изменяется независимо.

Логика алгоритма языко-независима и работает над абстрактным представлением программы, описываемым модулем

```
module AbstractRepr =
  struct

  type nt =
    (* Операция взятия адреса *)
    | Mem of nt
    (* Разыменовывание указателя *)
    | Deref of nt
    (* Динамическое выделение памяти с помощью calloc *)
    | Dyn of string
    (* Динамическое выделение памяти с помощью malloc *)
    | Anon of string
    (* Абстракция для NULL *)
    | Null
    (* Неизвестное расположение в памяти *)
    | Any
    (* Имя переменной *)
    | Var of string

  type node =
      Start
```

```
    | Empty
    | Value of (( nt * nt) option) list * Cil.exp

  type edge = True | False | FallThrough | GoTo

end
```

Для реализации итерации по графу потока управления использовался алгоритм из библиотеки PRANLIB, параметризуемый описанным выше абстрактным представлением программы на следующей полурешетке:

```
module PointsToSemilattice =
  Semilattice.Make(
    struct
      type t = Relation.t

      let top = full
      let bottom = empty
      let equal x y = Relation.equal x y
      let cap x y = x <|> y
    end
  )
```

В качестве инициализирующей, используется функция $\tilde{S}$, описанная в (4.2). Берем описанную там же функцию перехода f. Они интерпретируются в диаграмму двоичных решений в соответствии с описанным в части 5 способом.

7 Заключение

В рамках дипломного проекта был предложен алгоритм потоко-зависимого анализа для языка C. Алгоритм основан на апроксимации выражений, влияющих на значения указателей во время выполнения программы и их последующей интерпретации в диаграммы двоичных решений.

В итоге разработаный анализ указателей был встроен в систему статического анализа $Creen$.

A Приложения

A.1 Пример

```
int main()
{
  int y,m;
  int* x;
  int* d;
  int* k;
  int** p;

  while (p == &k)
  {
    x = &y;
  };


  if (y == 0)
    x = malloc(4);
  else
    k = &m;

  p = &k;
  d = x;

  return 1;
}
```

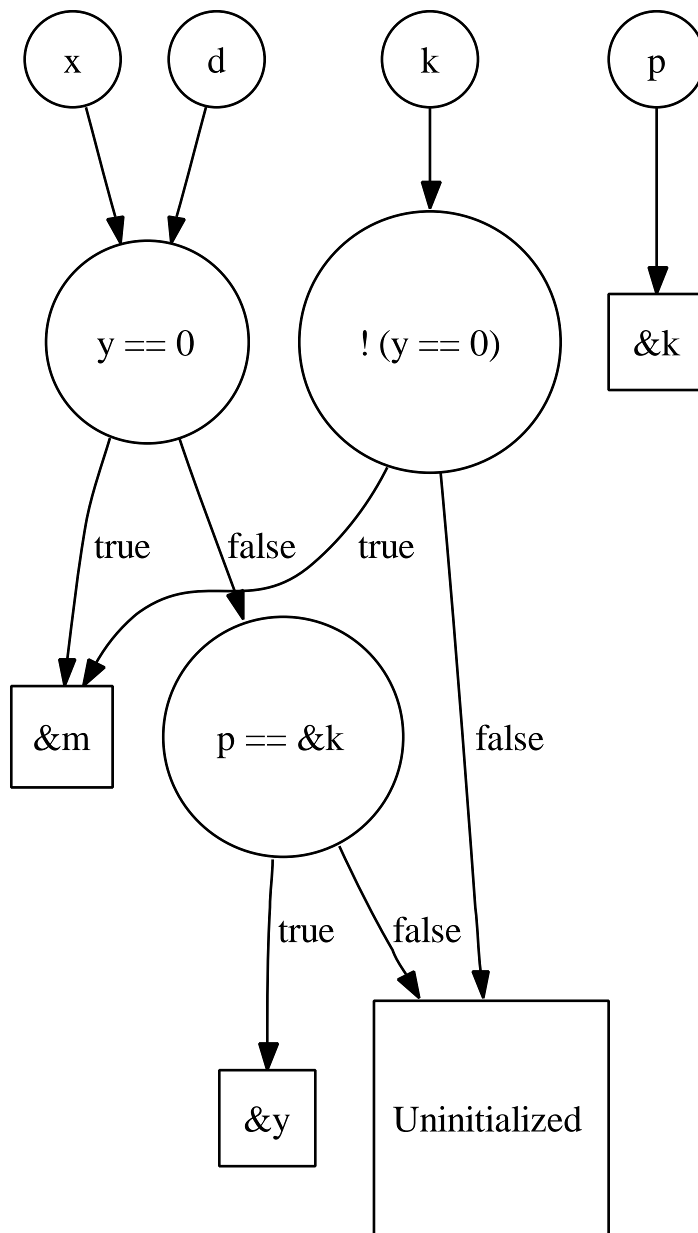

Рис. 2: PointsTo граф для примера

Список литературы

[А.Н76] Терехов А.Н. *Методы синтеза эффективной рабочей программы.* PhD thesis, Ленинградский ордена Ленина и ордена Трудового Красного Знамени Государственный Университет им. А.А. Жданова, Ленинград, 1976.

[Д.А06] Иванов Д.А. Реализация библиотеки диаграмм двоичных решений на языке ocaml. 2006.

[AA86] J. Ullman A. Aho, R. Sethi. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, New York, NY, USA, 1994. ACM Press.

[GCNW02] S.P. Rahul George C. Necula, Scott McPeak and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98: Proceedings*

of the 5th International Symposium on Static Analysis, pages 57–81, London, UK, 1998. Springer-Verlag.

[Lar89] James Richard Larus. *Restructuring symbolic programs for concurrent execution on multiprocessors*. PhD thesis, University of California, 1989. Chair-Paul Hifinger.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1992. ACM Press.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1997. ACM Press.

[Shi91] Olin Grigsby Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Pittsburgh, PA, USA, 1991.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.