Языково-независимый анализ указателей для библиотеки Pranlib

Л. Е. Чистов
lchistov@yandex.ru

При создании инструмента статического анализа или трансформации программ, учитывающего наличие в исходном языке указателей или ссылок, важно иметь способ получения информации об их возможных значениях. Для решения данной задачи было создано множество алгоритмов анализа указателей, однако их неотъемлемой характеристикой остается высокая сложность реализации и ориентация лишь на определенные языки программирования. В данной статье описан подход к созданию повторно используемых и языково-независимых статических анализаторов, основанный на введении промежуточного представления — языка Alias с недетерминированной семантикой. Кроме того, демонстрируется возможность решения задачи анализа указателей для языка C при помощи представленного подхода.

Введение

Анализ указателей — это статический анализ кода программ с общей памятью, позволяющий для каждого объекта, способного некоторым образом содержать ссылку на другой объект, построить описание множества возможных значений, которые могут быть приписаны ему во время исполнения программы.

Применение результатов анализа указателей повышает эффективность большинства использующихся в современных компилято-

рах оптимизаций. Помимо этого, анализ указателей также необходим в различных инструментах анализа кода, таких как проверка безопасности и верификация программ.

За последние 30 лет появилось более 80 статей, посвященных проблеме анализа указателей, однако можно с уверенностью сказать, что на данный момент не существует одного общепринятого алгоритма, достигающего оптимального соотношения точности, времени работы и простоты реализации. Одной из причин является появление новых языков и парадигм программирования. Техники анализа, разработанные для C-подобных языков, зачастую не дают удовлетворительного результат для объектно-ориентированных языков, таких как Java. Следует отметить также и объективную сложность проблемы: известно, что потоково-независимый точный анализ указателей является NP-трудной задачей [7], а задача точного потоково-зависимого анализа при использовании динамического выделения памяти алгоритмически неразрешима [10]. Данные результаты превращают задачу анализа указателей в поиск некоторого консервативного приближения к точному решению.

Реализация анализа указателей является сложной задачей, отнюдь не в каждом компиляторе используется нетривиальный анализ. Облегчить создание качественного компилятора могут специализированные системы и библиотеки статического анализа кода, одной из которых является библиотека анализа программ Pranlib [1], созданная на кафедре системного программирования математико-механического факультета СПбГУ.

В качестве языка реализации библиотеки был выбран функциональный язык программирования Objective CAML[1] по причине удобства функциональных языков для манипуляций с деревьями и графами и реальной применимости языка Objective CAML для создания программных продуктов.

Основным принципом Pranlib является независимость от конкретного языка программирования или представления, которая достигается через высокую степень общности и настраиваемости модулей и позволяет использовать библиотеку в качестве подсистемы произвольного компилятора или инструмента анализа кода. На

[1] http://caml.inria.fr

данный момент библиотека включает реализацию основных алгоритмов анализа потока управления, каркас для анализа зависимостей, «решатели» для общей задачи анализа потока данных и некоторые конкретные алгоритмы анализа потока данных — достигающие определения (reaching definitions), живые переменные (live variables) и др.

Для достижения необходимой степени общности анализа в рамках данной работы был создан язык описания операций над указателями Alias, который, с одной стороны, удобен для этой задачи, а с другой — имеет выразительную силу, достаточную для корректного отображения всех необходимых конструкций большинства современных высокоуровневых языков программирования, оперирующих объектами, по существу сходными со ссылками и указателями, в конструкции языка Alias.

В данной работе представлен внутрипроцедурный, т. е. применимый к отдельным процедурам и функциям, анализ указателей для библиотеки Pranlib. Анализ описан как сведение исходной задачи к общей задаче анализа потока данных.

1. Обзор существующих подходов

Алгоритмы, предложенные для решения задачи анализа указателей, можно классифицировать по разным характеристикам[2].

В зависимости от рода интересующей пользователя информации, выделяют несколько широких групп анализов: *анализ ссылок (alias analysis)*, *анализ указателей* в узком смысле (*pointer analysis*) и *анализ формы (shape analysis)*. Различие коренится в форме вопроса о структуре отношения указывания, заданного на элементах программы, таких как переменные и динамические объекты (грубо говоря, элементы x и y находятся в отношении указывания, если x содержит адрес y). В общем случае, анализ ссылок отвечает на вопрос о том, могут ли два пути доступа ссылаться на один и тот же объект. Анализ указателей для каждой переменной типа указатель (или, что по сути то же самое, объектной ссылки) определяет

[2]Обзор к решению задачи указателей в языке C можно найти, например, в [2].

множество возможных значений. Анализ формы используется для выяснения топологических свойств использующихся в программе структур данных. К примеру, нас может интересовать, задает ли отношение указывания ориентированный граф без циклов или дерево. Анализы формы являются наиболее специфическими и редко используемыми и не затрагиваются далее в этой статье.

Как и всякий статический анализ, анализ указателей предоставляет описание множества состояний программы. Под описанием можно понимать конечный набор свойств состояния, таких как «переменные A и B являются ссылками на один и тот же объект кучи» или «граф отношения указывания не содержит циклов». Поскольку в силу потенциальной бесконечности числа состояний программы в рамках анализа указателей не предоставляется возможным построить для каждого интересующего нас состояния в точности соответствующий ему набор свойств, ограничиваются приближением к нему. Согласно устоявшейся терминологии, must-анализ строит набор свойств не шире, а may-анализ — не у́же точного. Например, если нас интересует значение некоторой переменной A типа указатель, то принадлежность свойства «A указывает на B» результату must-анализа означает, что в каждом из описываемых состояний A указывает на B. Наоборот, если A указывает на B хотя бы в одном из описываемых состояний, то свойство «A указывает на B» принадлежит результату may-анализа.

По области применения все представленные в литературе алгоритмы делятся на *внутрипроцедурные* (*intra-procedural*) и *межпроцедурные* (*inter-procedural*). Внутрипроцедурные алгоритмы применимы для анализа отдельных процедур, функций или методов. При подобном анализе информация о воздействии остальных функций на состояние программы предполагается неизвестной. Встречающиеся в коде процедуры вызовы рассматриваются как обращения к черному ящику, чье поведение должно быть оценено консервативно таким образом, чтобы был учтён результат произвольных допустимых для него действий. Межпроцедурные алгоритмы используются для анализа всей программы в целом. В общем случае межпроцедурные алгоритмы значительно сложнее в реализации, точнее, медленнее и требуют больше памяти, чем внутрипроцедурные. Выбор области применения определяется конкретными задачами клиен-

та анализа; так, для проведения внутрипроцедурных оптимизаций часто бывает достаточно внутрипроцедурного анализа указателей.

1.1. Математические модели описания ссылочной информации

Одним из определяющих элементов в структуре алгоритма анализа указателей является выбранная модель описания ссылочной информации. Поскольку при наличии в языке операции динамического выделения памяти размер множества описываемых объектов потенциально неограничен, необходим адекватный способ консервативного описания ссылочной информации ограниченным набором свойств. Помимо этого, даже математически эквивалентные представления могут давать различные с точки зрения эффективности результаты.

Алгоритмы первой группы, так называемые *store-based* анализы, аппроксимируют состояние памяти программы во время выполнения, используя специальный граф, называемый *графом структуры памяти (Storage Shape Graph, SSG)*. Вершины графа соответствуют статическим и динамическим объектам программы, а дуги служат для представления отношения указывания. Для сохранения свойства ограниченности графа необходимо исходить из допущения, что одной «динамической» вершине может соответствовать потенциально неограниченное множество динамических объектов. Можно сказать, что все алгоритмы данной группы различаются выбором способа и момента для «слияния» динамических вершин. Простейший подход, описанный в [6], ориентирован на анализ локальных переменных и использует единственную абстрактную ячейку памяти (вершину SSG) для представления всей кучи. В статье [4] рассматривается базовый алгоритм, предполагающий слияние динамических объектов, созданных в одной и той же точке программы, и набор эвристик, позволяющих улучшить его характеристики. Подобный подход используется и в данной работе. Джонс и Мучник [9], в одной из первых работ по данной тематике, предлагают рассматривать одновременно множество возможных графов, впервые вводя понятие k-ограничения. Слиянию здесь подлежат динамические вершины, недостижимые из статических вершин по

пути длиной меньше k. Различные способы k-ограничения графов представлены в статье [8].

В основе второй группы алгоритмов лежит понятие *пути доступа* (*access path*). Для С-подобных языков путь доступа можно представить как пару из имени статического объекта (переменной) и, возможно пустой, последовательности операций разыменования указателя и доступа к полю структуры или объекта. Ланди и Райдер [11] описывают ссылочную информацию множеством всех тех пар путей доступа, которые могут ссылаться на один и тот же объект. Они также используют понятие k-ограничения, однако в смысле, отличном от указанного выше: используется консервативное предположение о том, что множеству эквивалентных путей доступа неявным образом принадлежат все пути с совпадающими префиксами, имеющими длину k.

1.2. Чувствительность к потоку управления

Важнейшей характеристикой внутрипроцедурных анализов указателей является чувствительность к потоку управления. *Потоково-зависимый* (*flow-sensitive*) анализ имеет дело с графом потока управления (ГПУ) процедуры — ориентированным графом, вершины которого соответствуют операторам процедуры, а ребра — переходам между ними. Результатом анализа является приписанное каждой вершине графа описание ссылочной информации, конкретная форма которого зависит от выбранной модели. Описание должно быть корректно для произвольного состояния, в котором программа может оказаться в процессе исполнения, непосредственно до (или, в зависимости от конкретного подхода, непосредственно после) выполнения оператора, приписанного к данной вершине. Чувствительность заключается в том, что учитываются только те состояния, которые могут быть получены в результате прохода программы по пути в графе потока управления. Все алгоритмы поиска подобной «корректной» разметки графа явным или неявным образом представляют собой сведение исходной задачи к общей задаче анализа потока данных (*см. разд. 2.*).

Потоково-независимые(*flow-insensitive*) анализы не используют информацию о порядке операторов в процедуре, рассматри-

вая неупорядоченные наборы тех операторов, чьё исполнение может привести к модификации отношения указывания. При этом используется консервативное предположение о том, что операторы из набора могут быть выполнены произвольное количество раз и в произвольном порядке. В общем случае можно утверждать, что потоково-независимые анализы обеспечивают ощутимо меньшую точность результата при значительном выигрыше в скорости работы по сравнению с потоково-зависимыми анализами. К примеру, алгоритм Штеенсгарда [13] с временной сложностью $O(N * \alpha(N, N))$, где α — обратная функция Аккермана, а N — размер программы, является самым быстрым нетривиальным алгоритмом анализа указателей, однако не обеспечивает удовлетворительной точности.

На сложность анализа указателей оказывает существенное влияние адресная арифметика, приведение типов, агрегированные структуры данных, механизмы определения локальных синонимов, такие как объединения (union) в языке C. Чрезмерно консервативная обработка подобных конструкций может существенно понизить точность анализа. Отдельными областями, не затронутыми в представленном обзоре, являются анализ языков уровня ассемблера и межпроцедурные алгоритмы анализа.

Можно выделить два основных класса метрик, применяемых для оценки точности алгоритмов. Авторы метрик первого класса рассматривают реализации алгоритмов анализа указателей как самодостаточные модули и используют средние количественные характеристики представлений ссылочной информации: размер графа структуры памяти, количество пар путей доступа, ссылающихся на один и тот же объект, и т. п. Метрики второго класса основаны на использовании результатов анализа в некоторой клиентской программе (например, оптимизирующем компиляторе) и оценке их влияния на его работу (например, количество проведенных оптимизаций как с использованием результатов анализа, так и без него).

2. Общая задача анализа потока данных

Предлагаемый в данной работе подход использует сведение задачи анализа указателей к общей задаче анализа потока данных. Мы дадим описание формальной постановки задачи, ориентируясь в своем изложении на монографию Нильсена [12].

Пусть имеется программа S, представляющая собой набор операторов, помеченных метками из множества $Label$. Нас интересует получение статической информации о состоянии программы в каждой ее точке. Для представления информации вводится полурешетка свойств — $(L, \sqcap)$; каждый элемент множества L соответствует некоторому свойству состояния программы, а операция $\sqcap$ — построению суммарного свойства.

Определение 1. *Пара $(L, \sqcap)$ называется* полурешеткой, *если операция $\sqcap : L^2 \to L$ обладает следующими свойствами:*

- $x \sqcap x = x$ — *идемпотентность;*
- $x \sqcap y = y \sqcap x$ — *коммутативность;*
- $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ — *ассоциативность.*

Определение 2. *Четверка $(L, \sqcap, \bot, \top)$ называется* ограниченной полурешеткой, *если:*

- *$(L, \sqcap)$ — полурешетка;*
- $\bot \in L \wedge \forall \ell \in L : \ell \sqcap \bot = \bot$;
- $\top \in L \wedge \forall \ell \in L : \ell \sqcap \top = \ell$.

Операция $\sqcap$ индуцирует на множестве L отношение частичного порядка: $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$. Отношение $\sqsubseteq$ интерпретируется как включение одного свойства другим; так, $x \sqsubseteq y$ означает $x \Rightarrow y$.

Важной с точки зрения практической реализации характеристикой является конечность итеративного процесса уточнения свойства.

Определение 3. *$(L, \sqcap)$ — полурешетка с обрывающимися цепями, если $\forall x_1 \sqsupseteq x_2 \sqsupseteq ... \; \exists N \; \forall n \geq N \; x_n = x_{n+1}$.*

Определение 4. *Пусть L — полурешетка. Высотой полурешетки $h(L)$ называется максимальная длина убывающей цепи.*

Изменение свойств состояния программы под действием операторов описывается элементами множества функций перехода $\mathcal{F}$. Существенным требованием, предъявляемым к функциям перехода в задачах анализа потока данных, является их монотонность.

Определение 5. *Функция $f : L \to L$ называется монотонной, если $\forall \ell, \ell' \in L \ (\ell \sqsubseteq \ell' \Rightarrow f(\ell) \sqsubseteq f(\ell'))$.*

Если функции перехода описывают изменение информации при переходе через точки программы, то *поток* задает пути ее распространения. Формально говоря, поток F есть множество упорядоченных пар $(l, l') \in Label \times Label$. В качестве примеров потоков, наиболее часто использующихся на практике, можно привести прямой поток $flow(S)$, представляющий собой множество ребер в графе потока управления G программы S, и обратный поток $flow_R(S)$, соответствующий графу, получающемуся в результате инвертирования ребер графа G. Анализы, определяемые на основе подобных потоков, называются, соответственно, прямым и обратным анализом.

Далее на некотором подмножестве *экстремальных* меток программы задаются граничные условия (элементы полурешетки свойств), соответствующие некоторым априори известным свойствам программы в определенных ее точках.

Теперь мы готовы к тому, чтобы сформулировать задачу анализа потока данных.

Определение 6. *Задачей анализа потока данных для программы S называется шестерка $(L, \mathcal{F}, F, E, \iota, f.)$, где:*

- *L является ограниченной полурешеткой свойств с обрывающимися цепями;*
- *$\mathcal{F}$ — это пространство монотонных функций перехода, замкнутое относительно функциональной композиции и содержащее тождественную функцию id;*
- *F — конечный поток;*
- *E — множество экстремальных меток;*

- ι — *граничные условия;*
- $f.$ — *отображение, сопоставляющее меткам программы* S *функции перехода.*

Под процессом решения задачи анализа потока данных обычно понимают построение разметки операторов программы элементами полурешетки, в некотором смысле корректно описывающей состояние программы в каждой ее точке. Мы будем искать разметку M в виде пары функций:

$$(M_\circ, M_\bullet) \in (Label \to L) \times (Label \to L).$$

Решением задачи анализа потока данных называется неподвижная точка системы уравнений:

$$\begin{aligned} \mathrm{M}_\circ(l) =& \bigsqcap\{\mathrm{M}_\bullet(l') \mid (l', l) \in F\} \sqcup \iota_E^l, \\ & \text{где } \iota_E^l = \begin{cases} \iota, & l \in E \\ \top, & l \notin E \end{cases}; \\ \mathrm{M}_\bullet(l) =& f_l(\mathrm{M}_\circ(l)). \end{aligned} \tag{1}$$

Описанный в [12] алгоритм с рабочим списком позволяет получить наибольшую (соответствующую наиболее точному решению) неподвижную точку системы (1). Для полурешетки с конечной высотой время работы алгоритма составляет $O(|F| * h(L))$.

3. Язык Alias

Язык Alias был разработан с целью создания унифицированного представления для описания действий с указателями и ссылками в высокоуровневых языках программирования, над которым было бы достаточно удобно проводить анализ указателей. В данном разделе представлены синтаксис и семантика языка Alias.

Следует отметить, что Alias не является полноценным языком программирования, в частности потому, что содержит некоторое число специфических операций, результат которых не известен достоверно до их исполнения (не только статически, но и во время выполнения программы), но является элементом некоторого множества возможных результатов. Одним из следствий наличия подобных операций является недетерминированность семантики языка. Они были включены в язык для консервативного описания

тех конструкций языков программирования, которые не являются достаточно общими, и операций, введение которых в качестве детерминированных исключительно усложнило бы анализ указателей.

3.1. Модель памяти

Далее по тексту мы будем пользоваться следующим условным обозначением: $\overline{X} = X \cup \{undef\}$.

Основным объектом предлагаемого анализа является *блок* памяти, представляющий собой аналог понятия переменной, использующегося в императивных языках программирования. Множество всех блоков, обозначающееся далее по тексту как *Block*, наряду с отношением «$\in$» на нем, формирует лес, причем порядок, в котором перечисляются потомки вершины, считается существенным. Будем говорить, что блок b является непосредственным подблоком блока b', если $b \in b'$. Определим отношение «$\in_*$» как рефлексивно-транзитивное замыкание «$\in$» и будем говорить, что блок b является подблоком блока b', если $b \in_* b'$.

Выделяется множество *Simple* всех «листовых» блоков. Каждый «листовой» или, иначе говоря, *простой блок* может быть понят как неделимая ячейка памяти, а каждый *сложный блок* (т. е. не являющийся простым) — как некоторый контейнер или структура, состоящая из множества подблоков.

Каждому простому блоку в процессе выполнения программы сопоставляется значение из множества $\overline{Block}$, т. е. блок как ячейка памяти может содержать либо ссылку на другой блок, либо специальное значение, соответствующее неопределенности.

Наряду со множеством блоков рассматривается ориентированный ациклический граф (дэг) *регионов свойств*. Каждый элемент множества регионов Region соответствует определенному свойству расположенных в нем блоков. Отношение принадлежности блока b региону r или, что тоже самое, обладание блоком b свойством r мы будем обозначать как $b \lhd r$. Отношение «$<:_1$», выбранное таким образом, чтобы вместе со множеством Region задавать ациклический ориентированный граф, для каждого региона r определяет множество его непосредственных подрегионов $\{r' | r' <:_1 r\}$.

Отношение вложенности «<:» определяется как рефлексивно-транзитивное замыкание отношения «$<:_1$». Вложенность региона r в регион r' ($r <: r'$) можно интерпретировать как уточнение свойством r свойства r'. Блок может принадлежать более чем одному региону, причем выполняются следующие соотношения:

- $r <: r' \wedge b \lhd r \Rightarrow b \lhd r'$;
- $b \lhd r \wedge b \lhd r' \Rightarrow \exists r'' : (r'' <: r) \wedge (r'' <: r') \wedge (b \lhd r'')$.

В качестве возможных примеров свойств, описываемых регионами, можно привести типы обычных языков программирования и области размещения переменных в памяти (стек, куча, область глобальных переменных и т. д.)

Будем обозначать множество всех деревьев с заданным порядком на потомках, чьи вершины размечены элементами множества X, как $Tree_X$. Определим множество типов блоков *Type* как $Tree_{Region}$. Будем говорить, что блок b имеет тип t, где $t = r(t_1, .., t_n)$, если блок b принадлежит региону r и для любого i в промежутке от 1 до n его i-ый непосредственный подблок имеет тип t_i.

Отношение вложенности регионов может быть распространено на множество типов блоков следующим образом: $r(t_1, .., t_n) <: r'(t'_1, .., t'_n) \Leftrightarrow r <: r' \wedge \forall i\ t_i <: t'_i$.

Истинным типом блока b мы будем называть такой его тип t, который минимален с точки зрения отношения вложенности типов. Очевидным образом, у любого блока существует единственный истинный тип.

В целях простоты описания семантики мы будем предполагать, что множество *Block* является бесконечным и, более того, содержит бесконечное количество блоков каждого возможного типа.

3.2. Синтаксис языка Alias

В языке Alias имеются три вида синтаксических конструкций: *выражения* (*expr*), *операторы* (*stmt*) и *помеченные операторы*(*labeled*). Описание абстрактного синтаксиса языка представлено на рис. 1.

Программой на языке Alias называется упорядоченное множество помеченных операторов.

$$\begin{aligned} expr ::= \quad & block(Block) \mid new(Type) \mid sub(Subblock, expr) \mid \\ & value(expr) \mid region(expr) \mid some(Region) \mid any \\ stmt ::= \quad & expr = expr \mid black(Region^*, expr^*) \\ labeled ::= \quad & Label : stmt : Label^* \end{aligned}$$

Рис. 1. Абстрактный синтаксис языка

Множество $Subblock$ состоит из функций, представимых в виде композиции тождественной функции и функций вида «взять i-ый непосредственный подблок блока либо вернуть ошибочное значение, если подобного блока не существует».

Множество операторных пометок $Label$ может иметь произвольную природу. Программа P называется корректно размеченной, если в ее записи каждая пометка l встречается не более одного раза непосредственно перед символом «:», и для любой пометки l, расположенной справа от символа «:», помеченный оператор вида $l : s : \{l_i\}_{i=1}^n$ также принадлежит P. Далее мы будем рассматривать только корректно размеченные программы.

3.3. Семантика языка Alias

В процессе выполнения программы P ее состояние характеризуется множеством используемых ею (*активных*) блоков $B(P)$ и содержащимися в них значениями. $B(P)$ делится на множество начальных блоков $B_0(P)$, состоящее из блоков (и всех их подблоков), входящих в запись P, и множество блоков, динамически выделяенных в процессе выполнения программы.

Более формально, состояние программы P есть элемент множества:

$$State = \{(B, s) | B_0(P) \subseteq B \subset Block, s : Simple \cap B \to \overline{B}\}.$$

Мы определим последовательно семантику выражений, операторов и программ языка Alias. Отметим, что на каждом из указанных уровней семантика является недетерминированной. Формальное определение семантик всех уровней дано в [3].

3.3.1. Семантика выражений

Опишем семантику выражений неформально.

- Результатом вычисления выражения $block(b)$ является блок b.
- Конструкция new описывает динамическое выделение блока в процессе работы программы. Результатом вычисления выражения $new(t)$ в состоянии (B, s) является некий блок, обладающий истинным типом t и не принадлежащий множеству B; побочным эффектом вычисления является добавление блока вместе со всеми его подблоками ко множеству активных блоков B.
- Конструкция sub соответствует операции взятия подблока. Результатом вычисления выражения $sub(f, e)$ является применение функции f к результату вычисления e. Получение значения функции f, равного $error$, соответствует неприменимости подобной операции к некоторому блоку; в этому случае все выражение в целом считается не обладающим семантикой.
- Выражение $value(e)$ соответствует взятию значения блока (или разыменованию указателя); его результат есть содержимое блока, являющегося результатом вычисления e.
- Конструкция $region$ описывает операцию, возвращающую произвольный активный блок из региона, содержащего ее аргумент.
- Результатом вычисления выражения $some(r)$ может быть любой активный блок, обладающий свойством r.
- Результатом вычисления выражения any может быть произвольный активный блок либо значение $undef$.

Представленная семантика выражений языка Alias является недетерминированной; данное ее свойство обусловлено наличием конструкций $region$, $some$ и any, вычисление значений которых сводится к недетерминированной выборке блока из некоторого множества. Эти конструкции были введены в язык Alias для консервативного представления операций исходных языков, множество возможных значений которых лежит в некотором диапазоне. В качестве примеров подобных операций можно привести адресную арифметику и функции, возвращающие объекты определенного типа. В первом случае естественно сопоставлять областям памяти регионы,

а операциям прибавления целых чисел к указателю — выражение $region(..)$; во втором случае регионы могут соответствовать объектным типам, а выражение $some(..)$ — произвольному методу (без побочных эффектов).

3.3.2. Семантика операторов

Результатом выполнения оператора является модификация состояния программы. Опишем семантику операторов неформально.

- **Оператор записи «=»**. Запись в блок b_1 блока b_2 соответствует замене значения первого блока на ссылку на второй блок. Поскольку лишь простые блоки могут содержать ссылки, семантика присваивания в случае составного блока в левой части не определена.
- **Оператор черного ящика «black»**. Исполнение оператора $black(\{r_i\}_{i=1}^m, \{e_j\}_{j=1}^n)$ соответствует вызову функции. Набор r_i обозначает множество регионов, в некотором смысле *открытых* для функции, а набор e_i — множество выражений, чьи значения передаются функции в качестве фактических параметров.

  В основе принципа работы черного ящика лежат два взаимосвязанных понятия: *допустимая операция* и *видимый блок*.

  Имеются два вида допустимых операций. Операцией первого вида является создание блока произвольной структуры, каждый подблок которого содержится в открытом регионе. Операцией второго вида является запись ссылки на видимый для функции блок в видимый блок. Предполагается, что черный ящик выполняет произвольное конечное число операций. Блок считается видимым, если он *достижим* путем применения операций взятия подблока и взятия значения от блока, принадлежащего одному из регионов r_i либо являющегося значением одного из выражений e_i. Кроме того, все блоки, созданные в процессе работы черного ящика, также являются видимыми для него.

3.3.3. Семантика помеченных операторов и программ

В записи $l : S : L$ метка l служит для идентификации оператора S, в то время как метки перехода $l' \in L$ отсылают ко множеству операторов, чье выполнение потенциально может последовать за выполнением оператора S. По исполнении оператора S и изменении состояния программы под его действием, контроль недетерминированно переходит к одному из операторов программы, помеченных элементом множества L. Если множество меток перехода пусто, программа завершает исполнение.

Опишем множество расширенных состояний программы. Его элементами являются промежуточные состояния вида $\langle (B, s), l : S : L \rangle$ и финальные состояния вида $\langle (B, s) \rangle$. Отношение перехода от одного расширенного состояния к другому за один шаг «$\rightarrow$» фактически уже описано выше, за формальным определением мы как обычно отсылаем к работе [3]. Отношение перехода за произвольное число шагов «$\rightarrow^*$» определяется как рефлексивно-транзитивное замыкание «$\rightarrow$». Выделяется также начальное расширенное состояние:

$$ES_{start} = \langle (B_0, s_0),\ (l_1 : S_1 : L_1) \rangle\,,$$

где $B_0 = B_0(P)$ — начальное множество активных блоков, $s_0 = \lambda b.undef$ — начальное означивание блоков неопределенностью.

Выполненность отношения $ES_{start} \rightarrow^* (B, s)$ означает, что (B, s) является возможным результатом исполнения программы.

3.4. Пример программы на языке Alias

В качестве примера приведем программу на языке C (рис. 2) и соответствующую её функции `main()` программу на языке Alias (рис. 3). Начальными являются простые блоки x, a, b и сложный блок $\{a, b\}$, принадлежащие региону $Local$. Помимого этого, рассматривается изначально пустой регион $Heap$ (куча), блоки в котором создаются в процессе выполнения программы. Под функциями $first$ и $second$ мы понимаем функции взятия первого и второго подблока сложного блока.

Особого внимания заслуживает помеченный оператор 0. Он необходим в том случае, если функция $main$ может быть вызвана

```
int* g () { ... }

void main () {
  int **x;
  struct { int* a; int* b; } p;

  x = &p.a; //1

  if (p.a == 0) {
    x = x++; //2
  }
  else {
    p.b = g(); //3
  }

  *x = malloc (sizeof(int)); //4
}
```

Рис. 2. Программа на языке C

$$0 : black(Heap, Extern) : 1;$$
$$1 : x = sub(first, \{a, b\}) : 2, 3;$$
$$2 : x = region(value(x)) : 4;$$
$$3 : black(Heap, Extern) : 3';$$
$$3' : sub(second, \{a, b\}) = value(some(Heap)) : 4;$$
$$4 : value(x) = new(Heap);$$

Рис. 3. Программа на языке Alias

не только в качестве точки входа в приложение, но и из некоторого окружения — тогда операции, предваряющие ее вызов, должны быть консервативно описаны при помощи оператора черного ящика. Регион $Extern$ выделен для обозначения множества глобальных переменных, использующихся в этом окружении.

4. Алгоритм анализа указателей

В данном разделе представлено сведение задачи анализа указателей к общей задаче анализа потока данных.

4.1. Полурешетка свойств

По заданной программе $P = \{l_i : S_i : L_i\}_{i=1}^m$ мы будем строить полурешетку $L(P)$ следующим образом.

Определим множество *динамических блоков*:

$$Dynamic \subset Block \setminus B_0(P),$$

каждый из которых будет соответствовать набору (потенциально бесконечному) блоков, создаваемых в процессе работы программы. Мы будем «склеивать» в один динамический блок все блоки одинакового истинного типа, выделенные при помощи конструкции *new* при исполнении одного и того же оператора. Предполагается, что наряду с каждым сложным блоком, во множестве $Dynamic$ содержатся также все его подблоки.

Будем считать, что склейка задана частичной функцией:

$$alloc : Label \times Type \to Dynamic,$$

такой что $alloc(l, t)$ имеет истинный тип t (или не определено). Поскольку создание блоков может происходить не только явным образом, но и в процессе исполнения оператора черного ящика, для описания возможных значений подобных блоков вводится множество *псевдоблоков*:

$$Pseudo \subset Simple \setminus (B_0(P) \cup Dynamic).$$

Псевдоблок, непосредственно принадлежащий региону r, соответствует склейке всех блоков, созданных в этом регионе в результате исполнения операторов черного ящика. Соответствие задается частичной инъективной и сюръективной функцией:

$$pseudo : Region \to Pseudo.$$

Так как типы создаваемых в черных ящиках блоков не известны и количество типов не ограничено, будем предполагать, что псевдоблоки обладают всеми возможными типами одновременно. Формально, псевдоблоки являются простыми блоками, поэтому вместо отношения «$\in$» (определяющего обычные типы блоков) нам понадобится новое отношение «$\in_p$», заданное на множестве $Pseudo$:

$$pseudo(r_1) \in_p pseudo(r_2) \Leftrightarrow \exists black(R, \dots) \in P : r_1, r_2 \in R.$$

Легко заметить, что отношение «$\in_p$» индуцирует на множестве $Pseudo$ циклическую, а не древообразную структуру. В частности, каждый псевдоблок является своим собственным «подблоком».

Таким образом, множество блоков, оценку множества возможных значений которых необходимо выдать в качестве результата выполнения анализа, представимо в виде:

$$B^L(P) = B_0(P) \cup Pseudo \cup Dynamic.$$

Определение ограниченной полурешетки $L(P) = (L, \bot, \top, \sqcap)$ дано на рис. 4. Отметим, что $L(P)$, в силу конечности множества ее элементов, является полурешеткой с конечной высотой и, следовательно, полурешеткой с обрывающимися цепями.

$$L = Simple \cap B^L(P) \to 2^{\overline{B^L(P)}}$$
$$\forall b \ \bot(b) = \overline{B^L(P)}$$
$$\forall b \ \top(b) = \oslash$$
$$(\ell \sqcap h)(b) = \ell(b) \cup h(b)$$

Рис. 4. Полурешетка свойств

Предлагается следующая интерпретация элемента полурешетки: для $\ell \in L$ $\ell(b)$ есть множество возможных значений блока b (в некоторой точке выполнения программы); равенство $\ell(b) = B$ означает, что если блок b содержит ссылку на блок b', то $b' \in B$, и если значение b не определено, то $undef \in B$. В частности, из равенства $\ell(b) = \{b'\}$ следует, что блок b обязан указывать на блок b'.

4.2. Потоковые функции

Метке l программы P, соответствующей некоторому оператору S, мы должны сопоставить некоторую функцию g_l, описывающую преобразование элемента полурешетки под действием оператора S.

Нам потребуется вспомогательная функция:

$$val_l : expr \times L \to 2^{\overline{Block}}.$$

Формальное определение $val_l(e, \ell)$ и потоковых функций g_l дано в работе [3].

Функция $val_l(e, \ell)$ служит для построения консервативной оценки множества возможных значений выражения e, входящего в запись оператора, помеченного меткой l, в состоянии, описываемом элементом полурешетки ℓ:

- результатом вычисления выражения $block(b)$ является блок b, поэтому оценкой множества значений для данного выражения может служить $\{b\}$;
- поскольку все блоки одного истинного типа t, создаваемые в процессе исполнения оператора, помеченного меткой l, описываются одним динамическим блоком $alloc(l, t)$, то множество $\{alloc(l, t)\}$ является консервативной оценкой;
- оценка значения $sub(f, e)$ состоит из двух компонент: множества значений функции f на обычных блоках из $val_l(e, \ell)$ и множества всех подблоков (в смысле отношения «$\in_p$») псевдоблоков из $val_l(e, \ell)$;
- консервативной оценкой для выражения $value(e)$ служит множество всех возможных значений простых блоков, принадлежащих множеству значений выражения e;
- для построения консервативной оценки значения выражения $region(e)$ берутся все блоки из тех регионов, которым могут принадлежать значения выражения e;
- оценкой множества значений $some(r)$ является множество всех блоков, обладающих свойством r;
- поскольку выражение any может иметь произвольное значение, мы обязаны включить в него все блоки из множества $B^L(P)$ наряду с неопределенностью.

Неформально, потоковые функции можно описать следующим образом.

- Потоковая функция для оператора записи «$e_1 = e_2$» модифицирует элемент полурешетки, обновляя значения тех блоков, которые потенциально могут быть результатом вычисления выражения e_1. Различают два типа обновления.
  1) *Сильное* обновление происходит в том случае, когда лишь один простой блок может быть значением e_1, причем этот блок является начальным (принадлежит $B_0(P)$). В этом случае текущее значение данного блока в процессе выполнения программы с необходимостью будет заменено на ссылку на один из блоков, принадлежащих $val_l(e_2, \ell)$, поэтому данное множество служит консервативной оценкой множества возможных значений данного блока.
  2) *Слабое* обновление происходит тогда, когда мы не можем однозначно определить, в какой блок произойдет запись, из-за того, что количество простых блоков, которые могут возникнуть в левой части оператора, больше единицы, либо данный единственный блок является склейкой (динамическим или псевдоблоком). В результате слабого обновления ко множеству возможных значений каждого простого блока из $val_l(e_1, \ell)$ добавляется множество возможных значений выражения e_2.
- Действие потоковой функции для оператора черного ящика «$black(R, E)$» состоит в построении консервативной оценки B множества блоков, видимых для черного ящика, и добавления ко множеству значений каждого простого блока из B всего множества B, что соответствует допустимости записи произвольного видимого блока в произвольный видимый блок в определении семантики черного ящика. Консервативная оценка строится как замыкание множества блоков, которые принадлежат регионам R или могут быть результатами вычисления выражений E.

Несложно показать (см. [3]), что введенные потоковые функции обладают таким свойством, как монотонность.

4.3. Сведение задачи анализа указателей к задаче анализа потока данных

Пусть дана программа P $= \{l_i : S_i : L_i\}_{i=1}^{m}$. Построим по ней задачу анализа потока данных (L, $\mathcal{F}$, F, E, ι, g.) следующим образом:

- $L = L(P)$ (см. рис. 4);
- $\mathcal{F}$ — замыкание множества $\{id\} \cup \{g_l\}_{l \in Label(P)}$ относительно оператора функциональной композиции;
- F — прямой поток:

$$F = \{(l_i, l_j) \mid l_i : S_i : L_i, l_j : S_j : L_j \in P \wedge l_j \in L_i\};$$

- $E = l_1$;
- $\iota = \lambda b.\{undef\}$;
- $f_l = g_l$, где $l : S : L \in P$.

Решения представленной задачи анализа потока данных мы будем называть решениями задачи анализа указателей для программы P на языке Alias.

В работе [3] формально доказано, что произвольное решение является корректным в смысле описанной семантики языка Alias.

5. Реализация

Представленный алгоритм анализа указателей был реализован в качестве отдельного модуля AliasView библиотеки Pranlib.

Начнем описание реализации с введения некоторых понятий, относящихся к библиотеке Pranlib. Как было уже сказано ранее, библиотека содержит решатели общей задачи анализа потока данных. Анализ производится над так называемым *описанием программы* (модульный тип Program View), состоящим из *адаптера* (модульный тип Adapter), *абстрактора* (модульный тип Abstractor), графа потока управления программы (модульный тип CFG.Sig) и полурешетки свойств (модульный тип Semilattice.Sig). Под *представлением* (модульный тип ProgramView.Repr) понимается множество пометок вершин и ребер графа. ГПУ программы размечен элементами *конкретного* (некоторого произвольного пользовательского)

представления, тогда как анализ фактически производится над элементами *абстрактного* (приспособленного под некоторый анализ потока данных) представления. Отображение конкретного представления в абстрактное обеспечивается абстрактором. Адаптер содержит описание потоковых функций и начальной разметки элементами полурешетки.

Отметим, что за счет использования абстракторов мы добиваемся независимости анализов потока данных от пользовательских представлений программ.

Вписывается в данную систему понятий и анализ указателей. Абстрактным представлением в данном случае является язык Alias. Интерфейс модуля AliasView предоставляет функции для инициализации памяти программы (леса статических блоков и ациклического графа регионов свойств) и операторов языка. Основной функтор Analyse позволяет по начальному состоянию памяти, ГПУ некоторой процедуры исходной программы и абстрактору из произвольного пользовательского представления в Alias-представление построить модуль, содержащий результаты анализа указателей. Фактически, задача модуля AliasView сводится к построению полурешетки свойств и адаптера. Результат анализа описывается в виде пары функций may и must, определяющих по двум выражениям языка Alias, могут или должны ли пересекаться множества их значений.

Помимо этого, предусмотрена визуализация результатов анализа в виде кластеризованного графа. Кластеры верхнего уровня соответствуют вершинам графа потока управления программы и содержат графы отношения указывания в состоянии до и после выполнения оператора, приписанного к данной вершине. Визуализация реализована как отображение внутреннего представления результата анализа в программу на языке описания графов DOT[3], для которого существует средство автоматического перевода в графическое представление Graphviz[4].

В рамках проверки работоспособности анализа была использована система CIL[5], позволяющая по программе на языке C по-

[3] www.graphviz.org/doc/info/lang.html
[4] www.graphviz.org
[5] manju.cs.berkeley.edu/cil

лучить граф потока управления программы, размеченный операторами некоторого подмножества языка C под названием CIL (C Intermediate Language). Было произведено встраивание анализа указателей в систему CIL путем реализации абстрактора из CIL-представления в Alias-представление, что дало возможность произвести тестирование анализа на реальных программах на языке C. В качестве тестовых программ были выбраны стандартные утилиты unix-подобных систем: awk, grep, sed, patch.

Для верификации результатов анализа был реализован модуль для CIL, позволяющий инструментировать код программы утверждениями (asserts) о значениях переменных типа указатель. Использовалось два вида утверждений: позитивные (must) и негативные (no). Позитивные утверждения имеют вид: «значение указателя p есть адрес одной из переменных множества $X = \{x_i\}_{i=1}^{n}$». Негативные утверждения имеют вид: «значение указателя p не является адресом ни одной из переменных множества $X = \{x_i\}_{i=1}^{n}$». В таблице представлены количественные результаты инструментирования тестовых программ. Для каждой программы указано количество расставленных утверждений и средний размер множества X.

Результаты инструментирования программ

Project	Must	Must average	No	No average
gawk-3.1.7	71	1.09859154929577	324	7.4537037037037
grep-2.5	4	1	76	1.77631578947368
patch-2.6.1	21	1	6	5
sed-4.2	6	1	0	0

Процесс верификации анализа заключался в инструментировании данных программ утверждениями и автоматической проверке работоспособности результирующего кода на сопровождающих их стандартных тестовых наборах; показателем корректности было отсутствие ложных утверждений в процессе работы программы.

Количество выставленных утверждений может быть использовано в качестве метрики точности анализа. В частности, такая метрика может быть в будущем применена для сравнения различных версий алгоритма или для сравнения с иным алгоритмами.

Невысокие количественные показатели для большей части исследованных программ показали, что в своем нынешнем виде алгоритм, пожалуй, не очень полезен для анализа реальных программ,

```
void f () {
   struct { int *a; int b;} c;
   int** p;

   p = &c.a;
   *p = &c.b;
}
```

Рис. 5. Тестовый пример

в частности потому, что в них почти не создаются ссылки на статически распределенные области памяти, а динамически созданные области почти не используются в пределах непосредственно запрашивающих их процедур. Однако при этом открытым остается вопрос о том, насколько полезнее был бы гипотетический идеальный алгоритм анализа указателей.

На рис. 6 можно видеть визуализированные результаты анализа указателей для тривиальной программы с рис. 5, полученные с использованием системы CIL и библиотеки Pranlib. Жирные стрелки соответствуют отношению включения «$\in_*$», а пунктирные — отношению возможного указывания на блоках.

Заключение

Результатом данной работы являются описание алгоритма языково-независимого внутрипроцедурного анализа указателей и модуль библиотеки анализа программ Pranlib, реализующий представленный алгоритм.

С целью абстракции логики работы с указателями в языках программирования разработан специфический промежуточный язык Alias, над программами которого и производится анализ.

Предложенный алгоритм анализа указателей определен в качестве частного случая общего алгоритма анализа потока данных с соответствующей полурешеткой свойств и функциями перехода.

Произведено встраивание анализа в систему CIL, что дало возможность проверить достаточность выразительности языка Alias

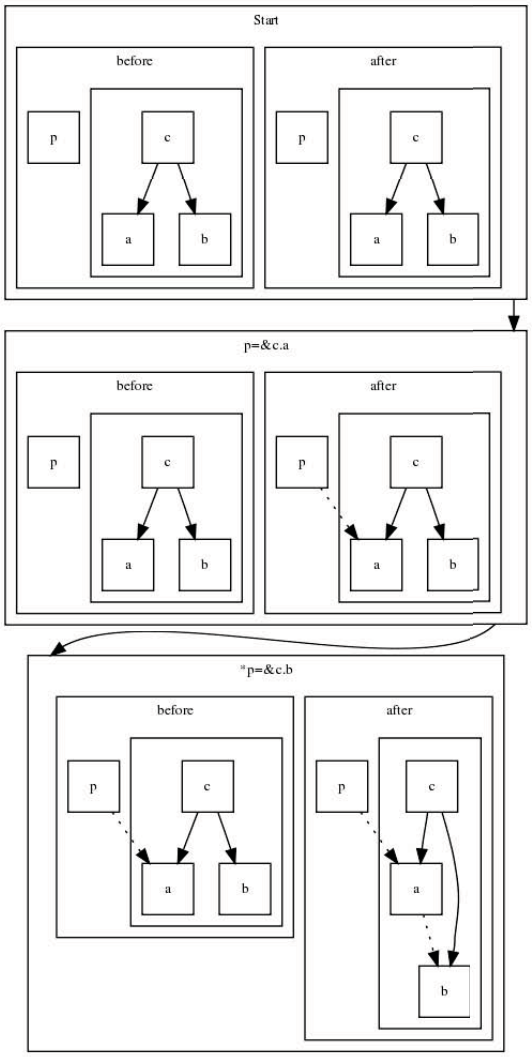

Рис. 6. Результат анализа указателей

для отображения в него конструкций языка C и протестировать анализ на реальных программах на языке C.

Реализовано графическое представление результатов анализа как отображение графа отношения указывания в программу на языке описания графов DOT.

Был разработан и реализован оригинальный метод верификации результатов анализа на основе инструментирования анализируемых программ утверждениями (asserts) о значениях указателей.

В качестве возможных путей развития анализа можно указать улучшение обработки динамически выделяемых блоков памяти и переход от внутрипроцедурного алгоритма к межпроцедурному.

Список литературы

[1] *Булычев Д. Ю.* Библиотека Pranlib. http://oops.tepkom.ru/projects/pranlib

[2] *Ицыксон В. М., Моисеев М. Ю., Ахин М. Х., Захаров А. В., Цесько В. А.* Алгоритмы анализа указателей для обнаружения дефектов в исходном коде программ. Системное программирование. Вып. 4: Сб. статей / Под ред. А. Н. Терехова, Д. Ю. Булычева. СПб.: Изд-во СПбГУ, 2009. С. 5–30.

[3] *Чистов Л. Е.* Языково-независимый анализ указателей для библиотеки Pranlib. Дипломная работа. СПбГУ, 2009. 50 с.

[4] *Chase D. R., Wegman M., Zadeck F. K.* Analysis of Pointers and Structures // Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. New York, United States, 1990. P. 296–310.

[5] *Choi J., Burke M., Carini P.* Efficient Flow-Sensitive Inter-Procedural Computation of Pointer-Induced Aliases and Side Effects // Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, Charleston, United States, January 1993. P. 233–245.

[6] *Emami M., Rakesh G., Hendren L. J.* Context-Sensitive Interprocedural Points-to Analysis in the Presense of Function Pointers // Proceedings of the ACM SIGPLAN 1994 Conference on Programming language design and implementation, Orlando, Florida, United States, June 1994. P. 242–256.

[7] *Horwitz S.* Precise Flow-Insensitive May-Alias Analysis is NP-hard // ACM Transactions on Programming Languages and Systems. Vol. 19, N 1, January 1997. P. 1–6.

[8] *Horwitz S., Pfeiffer P., Reps T.* Dependence Analysis for Pointer Variables // Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, 1989. P. 28–40.

[9] *Jones N. D., Muchnik S. S.* Program Flow Analysis, Theory and Applications. New Jersey: Prentice-Hall, 1981. 418 p.

[10] *Landi W.* Undecidability of Static Analysis // ACM Letters on Programming Languages and Systems. Vol. 13, N 4, 1996. P. 323–337.

[11] *Landi W., Ryder B. G.* A Safe Approximate Algorythm for Interpocedural Pointer Aliasing // ACM SIGPLAN Notices. Vol. 13, N 7. 1992. P. 235–248.

[12] *Neilson F., Nielson H. R., Hankin C.* Principles of Program Analysis. Berlin: Springer, 1999. 452 p.

[13] *Steensgaard B.* Points-to Analysis in Almost Linear Time // Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, United States, 1996. P. 32–41.