

САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

ЯЗЫКОВО-НЕЗАВИСИМЫЙ анализ
указателей для библиотеки Pranlib

Дипломная работа студента 544 группы

Чистова Леонида Ефимовича

Научный руководитель: к.ф.-м.н. Булычев Д.Ю.

Рецензент: к.ф.-м.н. Бульонков М.А.

“Допустить к защите”, Терехов А.Н.
зав.каф., д.ф.-м.н., проф.

Санкт-Петербург
2009

Оглавление

1	Введение	2
2	Обзор существующих подходов	4
2.1	Абстракции ссылочной информации	5
2.2	Чувствительность к потоку управления	7
2.3	Межпроцедурные алгоритмы	9
3	Общая задача анализа потока данных	11
4	Язык Alias	15
4.1	Модель памяти	15
4.2	Синтаксис языка Alias	17
4.3	Семантика языка Alias	18
4.3.1	Семантика выражений	18
4.3.2	Семантика операторов	20
4.3.3	Семантика помеченных операторов и программ	22
4.4	Пример программы на языке Alias	23
5	Предлагаемый анализ указателей	27
5.1	Полурешетка свойств	27
5.2	Потоковые функции	29
5.3	Сведение к задаче анализа потока данных	34
5.4	Корректность анализа	35
6	Реализация	43
7	Заключение	47

Глава 1

Введение

Анализ указателей есть 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.

Использование результатов анализа указателей повышает эффективность большинства использующихся в современных компиляторах 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 или даже нескольких общепринятых алгоритмов, покрывающих все требования и достигающих желаемого соотношения точности, времени работы и сложности реализации. Одной из причин является появление новых языков и парадигм программирования; техники анализа, разработанные для С-подобных языков зачастую не дают удовлетворительного результата 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 [1], а задача точного потоково-зависимого анализа в условиях наличия динамического выделения памяти алгоритмически неразрешима [2]. Данные результаты превращают задачу анализа указателей в поиск некоторого консервативного приближения к точному решению. В числе наиболее актуальных открытых вопросов, связанных с данной темой, можно назвать масштабируемость анализов, построение метрик для оценки точности, развитие инкрементальных анализов и анализов для систем Just-In-Time

компиляции [3].

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 Pranlib¹, созданная на кафедре системного программирования математико-механического факультета СПбГУ.

В качестве языка реализации библиотеки был выбран функциональный язык программирования Objective CAML² 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 данный момент библиотека включает реализацию основных алгоритмов анализа потока управления и решатели для общей задачи анализа потока данных (см. Гл. 3).

Поскольку для точного решения основных задач анализа потока данных во многих языках программирования не обойтись без предварительного анализа указателей, было принято решение разработать и реализовать анализ указателей для библиотеки Pranlib.

Для достижения необходимой степени общности анализа, в рамках данной дипломной работы, был создан язык описания операций над указателями Alias (см. Гл. 4), который, с одной стороны, достаточно удобен для проведения над ним анализа, а с другой имеет выразительную силу, достаточную для возможности корректного отображения всех необходимых конструкций произвольного языка программирования, оперирующего объектами, по существу сходными с 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 формально как сведение исходной задачи к общей задаче анализа потока данных; дано доказательство корректности алгоритма с точки зрения семантики языка Alias.

¹<http://oops.tepkom.ru/projects/pranlib>

²caml.inria.fr

Глава 2

Обзор существующих подходов

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

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 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.

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 построить для каждого интересующего нас состояния в точности соответствующий ему набор свойств не предоставляется возможным, ограничиваются приближением к нему. Согласно устоявшейся терминологии, *must-анализ* строит гарантированно более узкий, а *may-анализ* гарантированно более широкий набор свойств. 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 ” с необходимостью принадлежит результату may-анализа. Естественно считать относительно более точными may-анализ, результатом которого является меньший набор свойств и must-анализ, находящий больший набор свойств, однако в реальных ситуациях однозначно расположить алгоритмы по степени точности не представляется возможным.

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 точнее, значительно сложнее в реализации и медленнее внутрипроцедурных. Выбор области применения определяется конкретными задачами клиента анализа; так, для проведения внутрипроцедурных оптимизаций часто бывает достаточно внутрипроцедурного анализа указателей.

2.1 Абстракции ссылочной информации

Одним из определяющих элементов в структуре алгоритма анализа указателей является выбранная абстракция 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 [4], ориентирован на анализ локальных переменных и использует единственную абстрактную ячейку памяти (вершину SSG) для представления всей кучи. В статье [5] рассматривается базовый алгоритм, предполагающий слияние динамических объектов, созданных в одной и той же точке программы, и набор эвристик, позволяющих улучшить его характеристики. Подобный подход к абстракции ссылочной информации используется и в Главе 5 данной работы. Джонс и Мучник [6], 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 [7].

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 [8] 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 длиной не меньше k с совпадающими префиксами длины k . В работе [9] предложен способ компактного представления ссылочной информации в виде результата транзитивной редукции множества эквивалентных пар путей доступа.

Алгоритмы анализа формы концентрируются на топологических свойствах структур данных, хранящихся в памяти. Анализ, описанный в статье [10] использует одновременно три абстракции ссылочной информации: *атрибут формы* (*shape attribute*), *матрицу направлений* (*direction matrix*) D и *матрицу интерференции* (*interference matrix*) I . Атрибут формы для каждой переменной типа указатель определяет форму структуры данных, достижимой из этого указателя и принимает одно из трех значений: *дерево*, *дэг* и *произвольный граф*. Для двух перемен-

ных p и q типа указатель элемент матрицы $D[p, q]$ равен единице тогда и только тогда, когда область памяти, на которую указывает q , достижима из p . Элемент матрицы $I[p, q]$ равен единице тогда и только тогда, когда существует область памяти, достижимая из p и q . По сути, обе матрицы являются вспомогательными структурами и служат для более точного вычисления атрибута формы.

2.2 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 абстракции. 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. Главу 3), путем определения операции слияния и функций перехода. Операция слияния соответствуют совмещению нескольких описаний ссылочной информации в вершинах с более чем одним предшественником. Функции перехода вычисляют результаты воздействия исполнения операторов на описания.

Потоково-независимые (*flow-insensitive*) анализы не используют информацию о порядке операторов в процедуре, рассматривая неупорядоченные наборы тех операторов, чье 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.

При анализе языков с безопасной типизацией используются потоково-

независимые алгоритмы, основанные на системе типов. В [11] представлен трехуровневый анализ ссылок, с повышением точности анализа при переходе от уровня к уровню. На первом уровне предполагается, что каждая пара объектных ссылок, множества подтипов типов определений которых имеют непустое пересечение, может ссылаться на один и тот же объект памяти. На втором уровне учитывается структура путей доступа к объектам; так, два различных поля одного объекта, имеющие одинаковый тип, не могут соответствовать одной и той же области памяти. Третий уровень предполагает проход по всем операциям присваивания процедуры, с построением специального графа, вершины которого сопоставляются классам эквивалентности типов, а ребра — возможным отношениям указывания между объектами соответствующих типов.

Специфическая типовая система вводится в работе Штеенсгарда [12]. Каждой переменной и каждой точке выделения памяти сопоставляется тип, представляющий собой вершину специального так называемого *points-to* графа, абстрагирующего отношение указывания. Начальное сопоставление задается биективным отношением, однако в процессе работы алгоритма происходит слияние вершин графа, обеспечивающее поддержание инварианта I : “количество исходящих дуг для каждой вершины не превышает единицу”. Осуществляется проход по всем операторам присваивания процедуры, на каждом шаге которого происходит добавление в граф не более чем одной дуги, индуцированной присваиванием, и не более чем одной операций слияния (оценка количества операций следует из выполнения I). При формальном подходе каждому оператору присваивания сопоставляется ограничение, выражаемое равенством между парой типов, причем задача анализа указателей сводится к поиску минимального решения системы типовых уравнений. Алгоритм Штеенсгарда 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.

Близкий к описанному выше алгоритм, предложенный Андерсеном в [13], не предполагает слияния вершин *points-to* графа, абстрагирующего отношение указывания. Ключевым становится не отношение равенства, а отношение включения на типах: говорят, что тип A включает в себя тип B , если множество последователей вершины A содержит множество последователей вершины B . Таким образом, задача анализа указателей сводится к поиску минимального решения системы включений или, в другой формулировке, построению транзитивного замыкания динамически конструируемого графа, что подразумевает временную сложность $O(N^3)$.

В работе Шапиро и Хорвитц [14] представлены два алгоритма, развивающие изложенные выше идеи. Первый алгоритм, параметризованный натуральным числом k и некоторым разбиением переменных программы на k категорий, допускает слияние двух вершин *points-to* графа только в случае принадлежности соответствующих переменных к одной категории. Выбор предельных значений k , равных единице и числу переменных, дает алгоритмы Штеенгарда и Андерсена. Второй алгоритм использует первый в качестве подпроцедуры, вызывая его несколько раз для различных разбиений множества переменных на k категорий (число k остается параметром алгоритма) и находя пересечение множеств возможных значений указателей. Эвристика, использованная авторами статьи, заключается в выборе минимального множества разбиений, при котором каждые две переменные хотя бы в одном разбиении принадлежат различным категориям.

2.3 Межпроцедурные алгоритмы

Простейший подход к межпроцедурному анализу указателей, использованный в частности в [8, 5], представляет собой построение единого, так называемого *ICFG* (*Interprocedural CFG*), графа потока управления для всей программы, и применение к нему методов внутрипроцедурного анализа. Единственное отличие заключается в необходимости добавления специальных узлов для вызова и возврата из процедуры, в которых информация об указателях распространяется в обе стороны, что делает анализ двунаправленным. Понятно, что размеры *ICFG* графа, получаемого для реальной программы при подобном подходе, будут слишком велики для того, чтобы можно было говорить о какой-либо эффективности анализа.

В качестве шага вперед можно рассматривать предложенное в [9] выделение двух уровней итерации: уровня графа потока управления конкретной процедуры и уровня графа вызова процедур - *PCG* (*Procedure Call Graph*), вершины которого соответствуют процедурам программы, а ребра — вызовам.

Недостатком двух рассмотренных выше алгоритмов является полное отсутствие так называемой *чувствительности к контексту*.

Алгоритм межпроцедурного анализа потока данных называется чувствительным к контексту, если он некоторым образом разделяет информацию относящуюся к различным путям исполнения программы. Поскольку при наличии рекурсивных функций множество возможных путей исполнения является потенциально бесконечным, слияние информа-

ции необходимо; поэтому следует говорить скорее о степени чувствительности, нежели чем об абсолютном обладании данным свойством. Чувствительность к контексту необходима, в частности, для исключения из рассмотрения нереализуемых путей вызовов, то есть путей в графе потока управления, по которым программа не может пройти в процессе исполнения (например, в которых точка вызова некоторой процедуры не совпадает с точкой возврата).

В [4] было предложено повышать степень чувствительности путем клонирования нерекурсивных процедур; причем создается столько экземпляров процедуры, сколько существует путей в PCG от стартовой процедуры, что может повлечь за собой экспоненциальный взрыв. Для борьбы с рекурсивными процедурами были разработаны методики аппроксимации потенциально бесконечных путей вызовов. *n*-CFA подход предполагает дифференциацию тех контекстов, у которых не совпадают наборы из *n* последних точек вызовов. Более тонкие алгоритмы допускают лишь слияние информации, соответствующей двум контекстам, различающимся только числом итераций по рекурсивным циклам в PCG.

Следует также отметить, что задача межпроцедурного анализа указателей значительно усложняется при наличии в языке виртуальных функций или указателей на функции, так как в этом случае граф вызовов процедур не может быть полностью построен во время компиляции, а чересчур консервативные предположения о его структуре приводят к ухудшению точности анализа.

На сложность анализа указателей оказывают определенное влияние также такие элементы анализируемых языков, как наличие адресной арифметики, приведения типов, агрегированных структур данных, присутствие способа задания локальных синонимов, таких как объединения (`union`) в C-подобных языках. Чрезмерная консервативность в обработке этих особенностей может существенно сказаться на точности анализа. Отдельной областью, не затронутой в представленном обзоре, является анализ языков уровня ассемблера.

Глава 3

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 [15], посвященную различным подходам к анализу программ.

Пусть имеется программа S , представляющая собой набор операторов, помеченных метками из множества $Label$. Предметом нашего интереса является получение статической информации о состоянии программы в каждой ее точке. Для представления информации вводится так называемая 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 $\sqcap : L^2 \rightarrow L$ 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 $(L, \sqcap, \perp, \top)$ называется ограниченной полурешеткой, если:

- $(L, \sqcap)$ - полурешетка;

- $\perp \in L \wedge \forall \ell \in L : \ell \sqcap \perp = \perp$;
- $\top \in L \wedge \forall \ell \in L : \ell \sqcap \top = \ell$.

Операция $\sqcap$ индуцирует на множестве L отношение частичного порядка: $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$. В зависимости от постановки исходной задачи, отношение $\sqsubseteq$ зачастую интерпретируют как отношение включения одного свойства другим; так, $x \sqsubseteq y$ может означать $x \Rightarrow y$. При подобном подходе важной с точки зрения практической реализации характеристикой становится конечность итеративного процесса уточнения свойства:

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

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 виду функций перехода в постановке задачи анализа потока данных, является их монотонность.

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

Если функции перехода описывают трансформацию 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 используемых 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. К примеру, для прямого анализа в качестве множества экстремальных меток может быть выбрана метка, сопоставленная входу в программу, а в качестве граничного условия — свойство начального состояния программы.

Теперь мы готовы к тому, чтобы сформулировать постановку задачи анализа потока данных:

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

- L — ограниченная полурешетка свойств с обрывающимися цепями;
- $\mathcal{F}$ — пространство монотонных функций перехода, замкнутое относительно функциональной композиции и содержащее тождественную функцию id ;
- F — конечный поток;
- E — множество экстремальных меток;
- ι — граничное условие (значение, приписываемое экстремальным меткам);
- $f.$ — отображение, сопоставляющее меткам программы S функции перехода.

Под решением 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 $(M_\circ, M_\bullet) \in (Label \rightarrow L) \times (Label \rightarrow L)$.

Среди корректных решений задачи анализа потока данных можно выделить более и менее точные. Отношение $\sqsubseteq$ естественным образом индуцирует порядок на решениях:

$$M^1 \sqsubseteq M^2 \Leftrightarrow \forall l \ M_\circ^1(l) \sqsubseteq M_\circ^2(l) \wedge M_\bullet^1(l) \sqsubseteq M_\bullet^2(l).$$

Для описания точного решения задачи анализа потока данных нам потребуется понятие пути. Неформально говоря, путь определяется как путь в графе, индуцированном потоком F , от некоторой экстремальной метки. Рассматриваются отдельно множество путей до метки, не включающих ее ($path_\circ$) и множество путей до метки, включающих ее саму ($path_\bullet$):

$$\begin{aligned} path_\circ(l) &= \{[l_1, \dots, l_{n-1}] \mid n \geq 1 \wedge \forall i < n: (l_i, l_{i+1}) \in F \wedge l_n = l \wedge l_1 \in E\} \\ path_\bullet(l) &= \{[l_1, \dots, l_n] \mid n \geq 1 \wedge \forall i < n: (l_i, l_{i+1}) \in F \wedge l_n = l \wedge l_1 \in E\} \end{aligned}$$

Действие, оказываемое операторами, принадлежащими пути, на состояние программы также описывается функциями перехода. Для пути $\vec{\ell} = [l_1 \dots l_n]$ функция перехода определяется как $f_{\vec{\ell}} = f_{l_n} \circ \dots \circ f_{l_1} \circ id$.

Определение 7. Точное решение задачи анализа потока данных (MOP Solution, Meet Over all Path solution):

$$\begin{aligned} MOP_{\circ}(l) &= \bigcap \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_{\circ}(l)\} \\ MOP_{\bullet}(l) &= \bigcap \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_{\bullet}(l)\} \end{aligned}$$

Хотя *MOP* решение представляется предпочтительным в большинстве ситуаций, в общем случае задача его нахождения является алгоритмически неразрешимой. Вследствие этого, задачу нахождения точного решения заменяют вычислением консервативного приближения к нему.

Приближенное решение ищется как неподвижная точка системы уравнений (*). Подробно описанный в литературе алгоритм с рабочим списком позволяет получить наибольшую неподвижную точку (*MFP*, *Maximal Fixed Point*) системы (*). Для случая полурешетки с конечной высотой время работы алгоритма составляет $O(|F| * h(L))$.

$$\begin{aligned} Analysis_{\circ}(l) &= \bigcap \{Analysis_{\bullet}(l') \mid (l', l) \in F\} \cup \iota_E^l \\ \text{где } \iota_E^l &= \begin{cases} \iota, & l \in E \\ \top, & l \notin E \end{cases} \quad (*) \\ Analysis_{\bullet}(l) &= f_l(Analysis_{\circ}(l)) \end{aligned}$$

Консервативность найденного приближения (при соответствующей интерпретации отношения $\sqsubseteq$) гарантируется соотношением $MFP \sqsubseteq MOP$.

Глава 4

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 данной главе 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.

4.1 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, обозначаемое 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

подблоком блока b' , если $b \in b'$.

Определим также отношение " $\in_*$ " на множестве $Block$ как рефлексивно-транзитивное замыкание отношения " $\in$ ".

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 $\overline{Block}$, то есть 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*. Элементы множества регионов $Region$ соответствуют определенным свойствам расположенных в них блоков. Отношение принадлежности блока b региону r , или, что тоже самое, обладания блоком b свойством r , мы будем обозначать как $b \triangleleft r$. 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:

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

Будем говорить, что блок b принадлежит непосредственно региону r , если r минимальный с точки зрения отношения вложенности регионов регион, содержащий b (обозначение $r = REG(b)$).

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,

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

Рис. 4.1: Абстрактный синтаксис языка

что блок b имеет тип t (обозначение: $b : t$), где $t = r(t_1, \dots, t_n)$, если блок b принадлежит региону r и его i -ый подблок имеет тип t_i .

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

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.

4.2 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. 4.1.

Программой на языке Alias называется упорядоченное множество помеченных операторов.

Множество

$$Subblock \subset Block \rightarrow Block \cup \{error\}$$

состоит из функций, представимых в виде композиции тождественной функции и функций вида “взять i -ый подблок блока либо вернуть ошибочное значение, если подобного блока не существуют”. Далее по тексту такие функции будут именоваться функциями взятия подблока.

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.

Множество меток программы P будем обозначать $Label(P)$.

4.3 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 выделяемых в процессе выполнения программы.

Более формально, состояние программы P есть элемент множества

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

Мы определим последовательно семантику выражений, операторов и программ языка Alias. Способ определения семантики языка в виде *систем переходов состояния* (state transition system) был взят из монографии [16]. Отметим, что на каждом из указанных уровней семантика является недетерминированной.

4.3.1 Семантика выражений

Поскольку вычисление выражения *new* меняет множество активных блоков программы, под значением выражения мы будем понимать пару

$$(B, b) \in 2^{Block} \times \overline{Block}$$

состоящую из обновленного множества активных блоков и результата вычисления выражения, представляющего собой блок либо неопределенность.

Будем говорить, что возможным значением выражения e в состоянии (B, s) является пара (B', b) , если $\langle (B, s), e \rangle \rightarrow_e (B', b)$. Определение отношения “ $\rightarrow_e$ ” представлено на Рис. 4.2.

Опишем семантику выражений неформально:

- Результатом вычисления выражения $block(b)$ является блок b ;

$$\langle (B, s), block(b) \rangle \rightarrow_e (B, b)$$

$$\langle (B, s), new(t) \rangle \rightarrow_e (B \cup \{b \mid b \in_* b'\}, b'), \text{ где } b' = next(B, t),$$

где $next$ — произвольное отображение из $Block \times Type$ в $Block$,
обладающее следующим свойством :

$\forall B, t$ блок $next(B, t)$ имеет истинный тип t

и ни он, ни один из его подблоков не содержится в B

$$\frac{\langle (B, s), e \rangle \rightarrow_e (B', b)}{\langle (B, s), sub(f, e) \rangle \rightarrow_e (B', f(b))}, \text{ если } b \in Block \wedge f(b) \in_* b$$

$$\frac{\langle (B, s), e \rangle \rightarrow_e (B', b)}{\langle (B, s), value(e) \rangle \rightarrow_e (B', s(b))}, \text{ если } b \in Simple$$

$$\frac{\langle (B, s), e \rangle \rightarrow_e (B', b)}{\langle (B, s), region(e) \rangle \rightarrow_e (B', b')}, \text{ если } b \in Block \wedge b' \in B' \wedge b' \triangleleft REG(b)$$

$$\langle (B, s), some(r) \rangle \rightarrow_e (B, b), \text{ где } b \in B \wedge b \triangleleft r$$

$$\langle (B, s), any \rangle \rightarrow_e (B, b), \text{ где } b \in \overline{B}$$

Рис. 4.2: Семантика выражений

- 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 , не принадлежащий множеству B . В качестве побочного эффекта вычисления можно рассматривать добавление блока (вместе со всеми 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 этом случае все выражение в целом считается не обладающим семантикой;
- Выражение $value(e)$ соответствует взятию значения блока (или разыменованию указателя). Его результат есть содержимое блока,

$$b \rightsquigarrow^s b' \Leftrightarrow \exists n \geq 0, \{b_i\}_{i=0}^n : (b_n = b' \wedge b_0 = b \wedge \forall i > 0 (b_i \in_* b_{i-1}) \vee b_i = s(b_{i-1})))$$

$$closure_s(B) = \{b' | \exists b \in B : b \rightsquigarrow^s b'\}$$

Рис. 4.3: Достижимость и замыкание

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. Данные 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 операции прибавления целых чисел к указателю выражение *region*($\cdot$); во втором регионы могут соответствовать объектным типам, а выражение *some*($\cdot$) произвольному методу (без побочных эффектов).

4.3.2 Семантика операторов

Результатом выполнения оператора является модификация состояния программы.

Эффект, оказываемый исполнением оператора на состояние программы, описывается отношением “ $\rightarrow_s$ ”, определение которого дано на Рис. 4.5.

Соответственно, запись $\langle (B, s), S \rangle \rightarrow_s (B', s')$ означает то, что под действием оператора S состояние (B, s) может перейти в состояние

$$\begin{aligned}
& R \vdash (B, s) \rightarrow_{black} (B, s) \\
& \frac{R \vdash (B, s) \rightarrow_{black} (B', s')}{R \vdash (B, s) \rightarrow_{black} (B', s'[b_1 \leftarrow b_2])}, \text{ где } b_1 \in B' \cap Simple, b_2 \in \overline{B'} \\
& \frac{R \vdash (B, s) \rightarrow_{black} (B', s')}{R \vdash (B, s) \rightarrow_{black} (B' \cup \{b \mid b \in_* b'\}, s')}, \text{ где } b' = next(B', t), t \in Tree_R
\end{aligned}$$

Рис. 4.4: Семантика недетерминированных действий черного ящика

$$\frac{\langle (B, s), e_1 \rangle \rightarrow_e (B', b_1) \quad \langle (B', s), e_2 \rangle \rightarrow_e (B'', b_2)}{\langle (B, s), e_1 = e_2 \rangle \rightarrow_s (B'', s[b_1 \leftarrow b_2])}, \text{ где } b_1 \in Simple$$

$$\begin{aligned}
& \langle (B, s), e_1 \rangle \rightarrow_e (B^{(1)}, b_1) \\
& \dots \\
& \langle (B^{(m-1)}, s), e_m \rangle \rightarrow_e (B^{(m)}, b_m) \\
& \frac{\{r_i\}_{i=1}^n \vdash (C, s) \rightarrow_{black} (B', s')}{\langle (B, s), black(\{r_i\}_{i=1}^n, \{e_i\}_{i=1}^m) \rangle \rightarrow_s (B', s')}, \text{ где} \\
& C = closure_s(\{b_i\}_{i=1}^m \cup \{b \mid \exists i : b \triangleleft r_i\})
\end{aligned}$$

Рис. 4.5: Семантика операторов

(B', s') .

Опишем семантику операторов более подробно:

- **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*.

$$statement(\{l_i : S_i : L_i\}_{i=1}^m)(l) = \begin{cases} l_j : S_j : L_j, & j \in [1..m] : l = l_j \\ undef, & \text{иначе} \end{cases}$$

Рис. 4.6: Определение функции *statement*

$$\frac{\langle (B, s), S \rangle \rightarrow_s (B', s')}{\langle (B, s), l : S : L \rangle \rightarrow \langle (B', s'), statement(P)(l') \rangle}, \text{ где } l' \in L$$

$$\frac{\langle (B, s), S \rangle \rightarrow_s (B', s')}{\langle (B, s), l : S : \emptyset \rangle \rightarrow \langle (B', s') \rangle}$$

Рис. 4.7: Семантика помеченных операторов программы *P*

Рассматриваются 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 видимой для функции блок в видимый блок. Предполагается, что черный ящик выполняет произвольное конечное число операций. Работа черного ящика описывается отношением “ $\rightarrow_{black}$ ”, формальное определение которого дано на Рис. 4.4. Запись $R \vdash (B, s) \rightarrow_{black} (B', s')$ означает, что под действием черного ящика при условии открытости регионов из множества R состояние (B, s) может перейти в состояние (B', s') .

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.

Формальное определение отношения достижимости “ $\rightsquigarrow^s$ ” и функции *closure_s*, позволяющей построить замыкание некоторого множества блоков относительно отношения достижимости, представлено на Рис. 4.3.

4.3.3 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 . Сразу по исполнении оператора S контроль недетерминированно переходит к одному из операторов программы,

помеченных элементом множества L . Если множество меток перехода пусто, программа завершает исполнение.

Для описания семантики программ нам понадобится функция выборки помеченного оператора по метке *statement*, определение которой дано на Рис. 4.6. Функция *statement* сопоставляет паре из программы и метки в ней оператор, помеченный данной меткой.

Теперь мы готовы к тому, чтобы определить семантику программ языка Alias в виде системы переходов.

Опишем множество расширенных состояний программы $EState$. 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 определено на Рис. 4.7. 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.

4.4 Пример программы на языке Alias

В качестве примера мы приведем программу на языке C (Рис. 4.8) и соответствующую её функции *main* программу на языке Alias (Рис. 4.9). Начальными являются простые блоки x , a , b и сложный блок $\{a, b\}$, принадлежащие региону *Local*. Помимо 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 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.

```

typedef struct
{
    int* a;
    int* b;
} pair;

int f ()
{
    ...
}

int* g ()
{
    ...
}

void main ()
{
    int **x;
    pair p;

    x = &p.a; //1

    if (f() == 1) //2
    {
        x = x++; //3
    }
    else
    {
        p.b = g(); //4
    }

    *x = malloc (sizeof(int)); //5
    **x = 121343; //6
}

```

Рис. 4.8: Программа на языке C

```

{
    0 : black(Heap, Extern) : 1;
    1 :  $x = \textit{sub}(\textit{first}, \{a, b\})$  : 2;
        2 : black(Heap) : 3, 4;
    3 :  $x = \textit{region}(\textit{value}(x))$  : 5;
        4 : black(Heap) : 4';
    4' :  $\textit{sub}(\textit{second}, \{a, b\}) = \textit{value}(\textit{some}(\textit{Heap}))$  : 5;
    5 :  $\textit{value}(x) = \textit{new}(\textit{Heap})$  : 6;
    6 :  $\textit{value}(\textit{value}(x)) = \textit{any}$  :
}

```

Рис. 4.9: Программа на языке Alias

Глава 5

Предлагаемый анализ указателей

В данной главе представлено предлагаемое сведение анализа указателей к общей задаче анализа потока данных (см. Гл. 3). Помимо этого, проведено доказательство корректности полученного анализа по отношению к введенной в Гл. 4 семантике языка Alias.

5.1 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

$$alloc : Label \times Type \rightarrow Dynamic$$

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*

$$\begin{aligned}
L &= Simple \cap B^L(P) \rightarrow 2^{\overline{B^L(P)}} \\
\forall b \perp(b) &= \overline{B^L(P)} \\
\forall b \top(b) &= \emptyset \\
(\ell \sqcap h)(b) &= \ell(b) \cup h(b)
\end{aligned}$$

Рис. 5.1: Полурешетка свойств

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

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

$$pseudo : Region \rightarrow Pseudo$$

Так как типы создаваемых в черных ящиках блоков неизвестны и количество типов неограниченно, мы, в некотором смысле, 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 $Pseudo$:

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

Рефлексивно-транзитивное замыкание отношения “ $\in_p$ ” обозначим как “ $\in_p^*$ ”.

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 предоставить в качестве результата выполнения анализа, представимо в виде

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

Определение ограниченной полурешетки $L(P) = (L, \perp, \top, \sqcap)$ дано на Рис. 5.1. Отметим, что $L(P)$ является полурешеткой с обрывающимися

цепями в силу конечности множества ее элементов. 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' .

5.2 Потокосые функции

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

Нам потребуются вспомогательные функции

$$val_l : expr \times L \rightarrow 2^{\overline{Block}} \text{ (см. Рис. 5.2)}$$

и

$$closure : L \times 2^{Block} \rightarrow 2^{Block} \text{ (см. Рис. 5.3)}$$

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 $alloc(l, t)$, постольку содержащее его множество 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 ;

$$\begin{aligned}
val_l(block(b), \ell) &= \{b\} \\
val_l(new(t), \ell) &= alloc(l, t) \\
val_l(sub(f, e), \ell) &= \{f(b) \mid b \in val_l(e, \ell) \setminus Pseudo \wedge f(b) \in_* b\} \cup \{b \mid b' \in val_l(e, \ell) \cap Pseudo \wedge b \in_p^* b'\} \\
val_l(value(e), \ell) &= \bigcup \ell(b), \quad b \in val_l(e, \ell) \cap Simple \\
val_l(region(e), \ell) &= \{b \in B^L(P) \mid \exists b' \in val_l(e, \ell), r = REG(b') : b \triangleleft r\} \\
val_l(some(r), \ell) &= \{b \in B^L(P) \mid b \triangleleft r\} \\
val_l(any, \ell) &= \overline{B^L(P)}
\end{aligned}$$

Рис. 5.2: Определение функции val

$$\begin{aligned}
b \rightsquigarrow^\ell b' &\Leftrightarrow \exists \{b_i\}_{i=0}^n, n \geq 0 : (b_n = b' \wedge b_0 = b \wedge \forall i > 0 (b_i \in_* b_{i-1}) \vee b_i \in \ell(b_{i-1})) \\
closure_\ell(B) &= \{b' \mid b \in B \wedge b \rightsquigarrow^\ell b'\}
\end{aligned}$$

Рис. 5.3: Определение функции $closure$

- 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.

Замечание: на данный момент консервативность оценок является по большей части декларативной, непосредственно следовать она будет из доказательства корректности анализа, приведенного в 5.4.

Функция $closure_\ell$ служит для построения консервативной оценки множества блоков, видимых для черного ящика, в состоянии, описываемом элементом полурешетки ℓ , и определяется аналогично функции $closure_s$ из Гл. 4.

$$\begin{aligned}
(f_l^{e_1=e_2}(\ell))(b) &= \begin{cases} val_l(e_2, \ell), & \{b\} = val_l(e_1, \ell) \cap Simple \setminus \{undef\} \wedge b \in B_0(P) \\ val_l(e_2, \ell) \cup l(b), & b \in val_l(e_1, \ell) \cap Simple \\ \ell(b), & \text{иначе} \end{cases} \\
(f_l^{black(R,E)}(\ell))(b) &= \begin{cases} B \cup \ell(b), & b \in B \cap Simple \wedge \\ & B = closure_\ell \left(\bigcup_{\substack{e \in E \\ b \in val_l(e, \ell)}} \ell(b) \cup \bigcup_{r \in R} \{b \mid b \triangleleft r, b \in B^L(P)\} \right) \\ \ell(b), & \text{иначе} \end{cases}
\end{aligned}$$

Рис. 5.4: Поточковые функции

Формальное определение потоковых функций дано на Рис. 5.4. Опишем их более подробно:

- Поточковая 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:
 - *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;
 - *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, принадлежащих регионам R или могущих быть результатами вычисления выражений E .

Покажем, что введенные потоковые функции обладают таким свойством, как монотонность.

Утверждение 1. Для любой метки l и выражения e , входящего в запись оператора, помеченного l , верно:

$$\ell_1 \sqsubseteq \ell_2 \Rightarrow val_l(e, \ell_1) \supseteq val_l(e, \ell_2)$$

Доказательство: Индукция по структуре выражения.

1. *База.* Значение функции val на простых выражениях $block$, new , $some$ и any не зависит от второго аргумента, что немедленно влечет за собой истинность доказываемого утверждения
2. *Индукционный переход.* Докажем утверждение для составного выражения $value$ (для выражений $region$ и sub доказательство проводится аналогично).

Пусть $\ell_1 \sqsubseteq \ell_2$ и $e = value(e')$ По индукционному предположению

$$val_l(e', \ell_1) \supseteq val_l(e', \ell_2) \quad (5.1)$$

Рассмотрим $b \in val_l(e, \ell_2)$. Из определения функции val следует, что

$$\exists b' \in val_l(e', \ell_2) \cap Simple : b \in \ell_2(b')$$

Из последнего и (5.1) выводим:

$$b' \in val_l(e', \ell_1) \cap Simple \quad (5.2)$$

Поскольку $\ell_1 \sqsubseteq \ell_2 \Rightarrow \ell_2(b') \subseteq \ell_1(b')$, верно:

$$b \in \ell_1(b') \quad (5.3)$$

По определению функции val , из (5.2) и (5.3) следует, что $b \in val_l(e, \ell_1)$. В силу произвольности b , взятого из $val_l(e, \ell_2)$, получаем: $val_l(e, \ell_1) \supseteq val_l(e, \ell_2)$.

□

Утверждение 2. Для любого помеченного оператора $l : S : \{l_i\}_{i=1}^n$ верно:

$$\ell_1 \sqsubseteq \ell_2 \Rightarrow f_l^S(\ell_1) \sqsubseteq f_l^S(\ell_2)$$

Доказательство. Достаточно показать, что

$$\forall b \in B^L(P) : f^S(\ell_1)(b) \supseteq f^S(\ell_2)(b)$$

1. Пусть оператор S имеет вид $e_1 = e_2$. Сравним значения функций $f_l^S(\ell_1)$ и $f_l^S(\ell_2)$ в различных точках.

а. пусть при вычислении $(f_l^S(\ell_2))(b)$ произошло сильное обновление блока $b \in B_0(P)$:

$$\{b\} = \text{val}_l(e_1, \ell_2) \cap \text{Simple} \setminus \{\text{undef}\}$$

тогда, по предыдущему утверждению,

$$\text{val}_l(e_2, \ell_2) \subseteq \text{val}_l(e_2, \ell_1)$$

и из того, что $b \in \text{val}_l(e_1, \ell_2)$, следует $b \in \text{val}_l(e_1, \ell_1)$, то есть значение блока b было обновлено и при вычислении $(f_l^S(\ell_1))(b)$:

$$f_l^{e_1=e_2}(\ell_1)(b) \supseteq \text{val}_l(e_2, \ell_1) \supseteq \text{val}_l(e_2, \ell_2) = f_l^{e_1=e_2}(\ell_2)(b)$$

Ч.т.д;

б. пусть произошло слабое обновление блока b :

$$b \in \text{val}_l(e_1, \ell_2) \cap \text{Simple}$$

аналогично предыдущему пункту, из

$$\text{val}_l(e_2, \ell_2) \subseteq \text{val}_l(e_2, \ell_1)$$

и

$$\ell_2(b) \subseteq \ell_1(b)$$

заключаем, что

$$f_l^{e_1=e_2}(\ell_1)(b) = \text{val}_l(e_2, \ell_1) \cup \ell_1(b) \supseteq \text{val}_l(e_2, \ell_2) \cup \ell_2(b) = f_l^{e_1=e_2}(\ell_2)(b)$$

Ч.т.д;

с. пусть значение блока b не было обновлено, т.е. $b \notin val_l(e_1, \ell_2)$. Поскольку нас интересуют только такие пары из оператора и элемента полурешетки, что под действием данного оператора состояние, описываемое данным элементом, переходит в некоторое корректное состояние программы, то множество $val_l(e_1, \ell_2)$ не может быть пустым. Из этого следует, что если произошло обновление значения блока b при вычислении $(f_l^S(\ell_1))(b)$, то это обновление было слабым, поскольку в этом случае верно:

$$val_l(e_1, \ell_1) \supseteq \{b\} \cup val_l(e_1, \ell_2)$$

отсюда мы можем заключить, что

$$f_l^{e_1=e_2}(\ell_1)(b) \supseteq \ell_1(b) \supseteq \ell_2(b) = f_l^{e_1=e_2}(\ell_2)(b)$$

ч.т.д

2. Пусть оператор S имеет вид $black(\{r_i\}_{i=1}^n, \{e_j\}_{j=1}^m)$.

Можно показать, что множество блоков, видимых для черного ящика, вычисляемое для элемента полурешетки ℓ_1 шире, чем для ℓ_2 , из чего следует истинность доказываемого утверждения.

□

5.3 Сведение 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 $(L, \mathcal{F}, F, E, \iota, g.)$ следующим образом:

- $L = L(P)$ (см. Рис. 5.1);
- $\mathcal{F}$ — замыкание множества $\{id\} \cup \{g_l\}_{l \in Label(P)}$ относительно оператора функциональной композиции;
- F — прямой поток:

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

- $E = l_1$;
- $\iota = \lambda b. \{undef\}$;

- $g_l = f_l^S$, где $l : S : L \in P$ (см. Рис. 5.4).

Точное решение представленной задачи мы будем считать точным решением, а произвольную разметку $M : M \subseteq MOP$ — приближенным решением задачи анализа указателей для программы P на языке Alias.

В параграфе 5.4 мы покажем, что точное решение является корректным в смысле описанной ранее семантики языка Alias; также, корректным будет и произвольное приближенное решение.

5.4 Корректность анализа

Элементы полурешетки $L(P)$ описывают состояния программы P , однако отнюдь не каждый элемент описывает корректно каждое состояние. Мы определим корректность описания формально как *согласованность*, подразумевающую существование отображения абстракции, устанавливающего взаимосвязь между потенциально бесконечным набором активных блоков программы и конечным множеством блоков полурешетки.

Определение 1. *Отображение $f : \bar{B} \rightarrow \overline{B^L(P)}$ назовем хорошим, если оно обладает свойствами*

1. $f|_{\overline{B_0(P)}} = id$ (тождественности на начальных блоках);
2. $\forall b \in B : REG(b) = REG(f(b))$ (сохранения истинного региона блока);
3. $\forall b \in B :$

$$f(b) \notin Pseudo \wedge g \in Subblock \wedge g(b) \neq error \Rightarrow f(g(b)) = g(f(b))$$

(сохранения структуры сложных блоков);

4. $\forall b, b' \in B : f(b) \in Pseudo \wedge b' \in_* b \Rightarrow f(b') \in Pseudo \wedge f(b') \in_p^* f(b)$
(сохранения отношения принадлежности для псевдоблоков).

Определение 2. $\ell \in L$ согласован с состоянием (B, s) (обозначение $\ell \models (B, s)$), если существует хорошее отображение абстракции

$$abs : \bar{B} \rightarrow \overline{B^L(P)}$$

такое, что

$$\forall b \in Simple \cap B \ abs(s(b)) \in \ell(abs(b))$$

Оказывается, что все элементы полурешетки, меньшие элемента, согласованного с некоторым состоянием, также согласованны с ним:

Утверждение 1. $\ell \models (B, s) \wedge \ell' \sqsubseteq \ell \Rightarrow \ell' \models (B, s)$.

Доказательство. Пусть $\ell' \sqsubseteq \ell$, $\ell \models (B, s)$. Тогда существует хорошее отображение abs , обеспечивающее согласованность ℓ и (B, s) . Для доказательства согласованности ℓ' возьмем то же самое abs : пусть

$$b \in Simple \cap B$$

тогда

$$abs(s(b)) \in \ell(abs(b)) \subseteq \ell'(abs(b))$$

□

Отталкиваясь от понятия согласованности, мы определим понятие корректности решения задачи анализа указателей (или, иначе, корректности разметки). Можно признать интуитивно очевидным следующее утверждение: разметка корректна, если для любого пути исполнения программы ее состояние в конце пути корректно описывается элементом разметки, приписанным к соответствующей метке программы. Для формализации приведенных выше размышлений нам понадобятся следующие условные обозначения, позволяющие сократить запись путей выполнения программы:

- $ES_i = \langle (B_i, s_i), l_{k_i} : S_{k_i} : L_{k_i} \rangle$ — некоторое промежуточное расширенное состояние;
- $ES_i^F = \langle (B_i, s_i) \rangle$ — некоторое финальное расширенное состояние.

Определение 3. Разметка $M = (M_o, M_\bullet)$ корректна, если:

1. для любого незавершенного пути исполнения $ES_{start} \rightarrow^* ES_n$ верно, что $M_o(l_{k_n}) \models (B_n, s_n)$;
2. для любого пути исполнения $ES_{start} \rightarrow^* ES_n \rightarrow ES_{n+1}^F$ верно, что $M_\bullet(l_{k_n}) \models (B_{n+1}, s_{n+1})$.

Переходам в семантике языка *Alias* мы сопоставили функции перехода анализа указателей. Покажем, что это сопоставление было произведено, в определенном смысле, корректно. Помимо свойства согласованности с состоянием, нас будет интересовать такая характеристика элементов полурешетки, как *сохранение неопределенности динамических блоков*.

Определение 4. $\ell \in L$ сохраняет неопределенность динамических блоков, если

$$\forall b \in \text{Dynamic} \cup \text{Pseudo} : \ell(b) \ni \text{undef}$$

Поскольку при создании блока в процессе выполнения программы его начальным значением является неопределенность, согласованно описывающий новое состояние элемент ℓ должен содержать *undef* в множестве возможных значений абстракции данного блока.

Множество элементов полурешетки, сохраняющих неопределенность динамических блоков будем обозначать как $\underline{L}$. Несложно проверить, что это свойство сохраняется при применении функций перехода.

Утверждение 2. *Вычисление значения выражения происходит согласованно: пусть (B, s) — состояние, $\ell \in \underline{L}$ — согласованно с (B, s) с помощью функции *abs*, l — метка, e — выражение, входящее в запись оператора, помеченного меткой l ,*

$$\langle (B, s), e \rangle \rightarrow_e (B', b)$$

и

$$\text{val}_l(e, \ell) = V$$

Тогда существует хорошее abs' — продолжение *abs* на B' :

$$\text{abs}'(b) \in V$$

Доказательство. Докажем истинность утверждения индукцией по структуре выражения e :

1. *База.* Пусть e — одно из простых выражений:

- (a) Если $e = \text{block}(b')$, то $B' = B$, $b = b'$, $V = \{b\}$ и в качестве abs' достаточно взять отображение *abs*.
- (b) Если $e = \text{new}(t)$, то

$$B' = B \cup \{b' \mid b' \in_* b\}$$

поэтому достаточно доопределить отображение абстракции *abs* для блока b и всех его подблоков. Сделаем это следующим образом: для произвольной функции взятия подблока g , такой, что $g(b) \neq \text{error}$, положим

$$\text{abs}'(g(b)) = g(\text{alloc}(l, t))$$

Очевидно, полученное отображение abs' будет хорошим.

- (с) Пусть $e = some(r)$. Возьмем abs в качестве abs' . Из определения семантики выражений следует, что $b \triangleleft r$, а из свойства произвольного хорошего отображения сохранять истинный регион блока — $abs(b) \triangleleft r$. Поскольку

$$V = \{b \in B^L(P) \mid b \triangleleft r\}$$

то $abs(b) \in V$.

- (d) Пусть $e = any$. Возьмем $abs' = abs$. Так как $V = B^L(P)$, верно следующее:

$$\forall b \ abs'(b) \in V$$

2. *Индукционный переход.* Пусть e — сложное выражение *value*, *sub* или *region*.

Из определения семантики выражений следует, что существует b' такой, что

$$\langle (B, s), e' \rangle \rightarrow_e (B', b')$$

Пусть

$$val_i(e', \ell) = V'$$

По индукционному предположению, существует хорошее abs' — продолжение abs на B' :

$$abs'(b') \in V'$$

Покажем, что $abs'(b) \in V$.

- (a) Пусть $e = value(e')$. Из определения семантики выражений следует, что

$$s(b') = b$$

Так как блок b' — простой, а хорошее отображение сохраняет свойство простоты, то $abs'(b')$ — также простой.

Если b' — старый блок, т.е. $b', s(b') \in \overline{B}$, то верно, что

$$abs'(b') = abs(b')$$

и

$$abs'(s(b')) = abs(s(b'))$$

следовательно

$$\ell(abs'(b')) = \ell(abs(b')) \ni abs(s(b')) = abs'(s(b'))$$

В противном случае, $s(b') = \text{undef}$ и соотношение

$$\ell(\text{abs}'(b')) \ni \text{abs}'(s(b'))$$

выполняется, поскольку ℓ сохраняет неопределенность динамических блоков. Отсюда

$$V = \bigcup_{b \in V' \cup \text{Simple}} l(b) \supseteq \ell(\text{abs}'(b')) \ni \text{abs}'(s(b')) = \text{abs}'(b)$$

- (b) Пусть $e = \text{sub}(f, e')$. Из определения семантики выражений следует, что

$$b = f(b')$$

Если $\text{abs}'(b') \in \text{Pseudo}$, то, поскольку отображение abs' — хорошее

$$\text{abs}'(b) \in_p^* \text{abs}'(b') \Rightarrow \text{abs}'(b) \in V$$

В противном случае

$$\text{abs}'(b) = \text{abs}'(f(b')) = f(\text{abs}'(b')) \in V$$

- (c) Доказательство для случая $e = \text{region}(e')$ проводится аналогично предыдущим пунктам, с использованием свойства хорошего отображения сохранять принадлежность региону.

□

Утверждение 3. *Исполнение оператора сохраняет согласованность: пусть $\ell \in \underline{L}$,*

$$\ell \models (B, s)$$

и

$$\langle (B, s), l : S : L \rangle \rightarrow \langle (B', s') \dots \rangle$$

тогда

$$g_l(\ell) \models (B', s')$$

Доказательство. Пусть ℓ согласованно с (B, s) с помощью функции abs и

$$\langle (B, s), l : S : L \rangle \rightarrow \langle (B', s') \dots \rangle$$

Из определения семантики помеченных операторов следует, что

$$\langle (B, s), S \rangle \rightarrow_s \langle (B', s') \rangle$$

Рассмотрим оператор S :

1. Пусть $S = e_1 = e_2$. Из определения операторной семантики следует, что существуют $b_1, b_2, B_1, B_2 = B'$:

$$\begin{aligned}\langle (B, s), e_1 \rangle &\rightarrow_e (B_1, b_1) \\ \langle (B_1, s), e_2 \rangle &\rightarrow_e (B_2, b_2) \\ s' &= s[b_1 \leftarrow b_2]\end{aligned}$$

Пусть

$$val_l(e_1, \ell) = V_1$$

и

$$val_l(e_2, \ell) = V_2$$

По утверждению 2 существует хорошее abs' :

$$abs'(b_1) \in V_1$$

и

$$abs'(b_2) \in V_2$$

Для доказательства согласованности $g_l(\ell)$ и (B', s') с помощью abs' рассмотрим блоки, значение которых могло измениться под действием функции перехода или исполнения оператора записи:

а. пусть

$$b \in B' \setminus B$$

тогда

$$s'(b) = \text{undef}$$

Поскольку ℓ сохраняет неопределенность динамических блоков, $g_l(\ell)$ также обладает данным свойством; отсюда

$$(g_l(\ell))(abs'(b)) \ni \text{undef} = s'(b) = abs'(s'(b))$$

б. пусть

$$b \in V_1 \cap Simple$$

и

$$b \neq b_1$$

В этом случае

$$|V_1| > 1$$

и

$$(g_l(\ell))(abs'(b)) \supseteq \ell(abs'(b)) = \ell(abs(b)) \ni abs(s(b)) = abs'(s'(b))$$

с. если $b = b_1$, то

$$(g_l(\ell))(abs'(b_1)) \supseteq V_2 \ni abs'(b_2) = abs'(s'(b_1))$$

2. Пусть $S = black(\{r_i\}_{i=1}^n, \{e_j\}_{j=1}^m)$. Из определения операторной семантики следует, что существуют B' , s' и $B^{(i)}, b_i$ для $i \in [1..m]$ (где $B^{(0)} = B$):

$$\langle (B^{(i-1)}, s), e_i \rangle \rightarrow_e \langle (B^{(i)}, b_i) \rangle$$

и

$$\{r_i\}_{i=1}^n \vdash (C, s) \rightarrow_{black} (B', s')$$

где

$$C = closure_s(\{b_i\}_{i=1}^m \cup \{b \mid \exists i : b \triangleleft r_i\})$$

Пусть

$$val_l(e_i, \ell) = V_i$$

По утверждению 2 существует хорошее abs' :

$$\forall i \ abs'(b_i) \in V_i$$

Расширим abs' на B' , сопоставив каждому блоку из $B'/B^{(m)}$, непосредственно принадлежащему региону r , псевдоблок $pseudo(r)$. Можно показать, что полученное подобным образом расширение также будет хорошим.

Далее, можно проверить, что множество видимых для черного ящика блоков в смысле полурешетки шире, чем подобное множество в смысле семантики, следствием из чего будет являться согласованность $g_l(\ell)$ с (B', s') .

□

Каждому пути выполнения программы соответствует путь в графе потока управления; при этом незавершенный путь вида

$$ES_{start} \rightarrow^* ES_n$$

переходит в элемент множества $path_o(l_{k_n})$, а завершенный путь вида

$$ES_{start} \rightarrow^* ES_n \rightarrow ES_{n+1}^F$$

в элемент множества $path_\bullet(l_{k_n})$. Покажем, что исполнение операторов программы на некотором пути также сохраняет согласованность.

Утверждение 4. Пусть $p = ES_{start} \rightarrow^* ES_n$ (или ES_n^F) — путь исполнения, $\vec{\ell}$ соответствующий ему путь в ГПУ, $\ell_0 \in \underline{L}$ и $\ell_0 \models (B_0, s_0)$, тогда

$$g_{\vec{\ell}}(\ell_0) \models (B_n, s_n)$$

Доказательство. Индукция по длине пути исполнения с использованием Утверждения 3. $\square$

Утверждение 5. Точное решение задачи анализа указателей корректно.

Доказательство.

1. $\iota \models (B_0, s_0)$: в качестве отображения abs достаточно взять тождественную функцию id . При этом, ι по определению сохраняет неопределенность динамических блоков.
2. Пусть $ES_{start} \rightarrow^* ES_n$ — незавершенный путь исполнения программы, $\vec{\ell}$ — соответствующий ему путь в ГПУ. По определению точного решения,

$$MOP_{\circ}(l_{k_n}) = \bigcap \{g_{\vec{p}}(\iota) \mid \vec{p} \in path_{\circ}(l_{k_n})\} \sqsubseteq g_{\vec{\ell}}(\iota) \models (B_n, s_n)$$

Следовательно,

$$MOP_{\circ}(l_{k_n}) \models (B_n, s_n)$$

3. Доказательство для случая завершенного пути проводится аналогично. $\square$

Утверждение 6. Произвольное приближенное решение задачи анализа указателей корректно.

Доказательство. Истинность утверждения является тривиальным следствием из утверждений 1 и 5. $\square$

Глава 6

Реализация

Описанный выше анализ указателей был реализован в рамках выполнения дипломной работы в качестве отдельного модуля AliasView библиотеки Pranlib.

Модуль AliasView производит сведение задачи анализа указателей к общей задаче анализа потока данных. Формально данное сведение было определено в предыдущей главе; ниже будет представлено его описание с точки зрения реализации.

Начнем 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. Можно привести следующий пример: пусть требуется решать задачи *Достигающие определения* (Reaching definitions, RD) и *Доступные выражения* (Avaible Expressions, AE) для программ на языках Java и C++. Тогда достаточно будет описать абстрактные пред-

ставления для задач RD и AE и четыре абстрактора ($\text{Java} \rightarrow \text{RD}$, $\text{Java} \rightarrow \text{AE}$, $\text{C++} \rightarrow \text{RD}$, $\text{C++} \rightarrow \text{AE}$) и решать задачи анализа потока данных только для абстрактных представлений.

Вписывается в данную систему понятий и анализ указателей. Абстрактным представлением в данном случае является язык Alias. Интерфейс модуля AliasView предоставляет функции для описания 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, по двум выражениям языка Alias определяющих, 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 DOT¹, для которого существует средство автоматического перевода в графическое представление Graphviz².

В рамках проверки работоспособности анализа была использована система CIL³, позволяющая по программе на языке C получить граф потока управления программы, размеченный операторами некоторого подмножества языка C (C Intermediate Language), эквивалентного ему по функциональности. 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.

Непосредственно проверить степень точности анализа и корректность его результатов на больших программах невозможно; для косвенной же оценки необходима реализация в библиотеке Pranlib использующих его результаты оптимизаций.

На Рис. 6.2 можно видеть визуализированные результаты анализа

¹www.graphviz.org/doc/info/lang.html

²www.graphviz.org

³manju.cs.berkeley.edu/cil

```

typedef struct
{
    int* a;
    int* b;
} pair;

void f ()
{
    int **x;
    int z;
    pair p;

    if (z == 1)
    {
        x = &p.a;
    }
    else
    {
        x = &p.b;
    }

    *x = &z;
}

```

Рис. 6.1: Тестовый пример

указателей для программы с Рис. 6.1, полученные с использованием системы CIL и библиотеки Pgranlib. Жирные стрелки соответствуют отношению включения “ $\in_*$ ”, а пунктирные — отношению возможного указания на блоках.

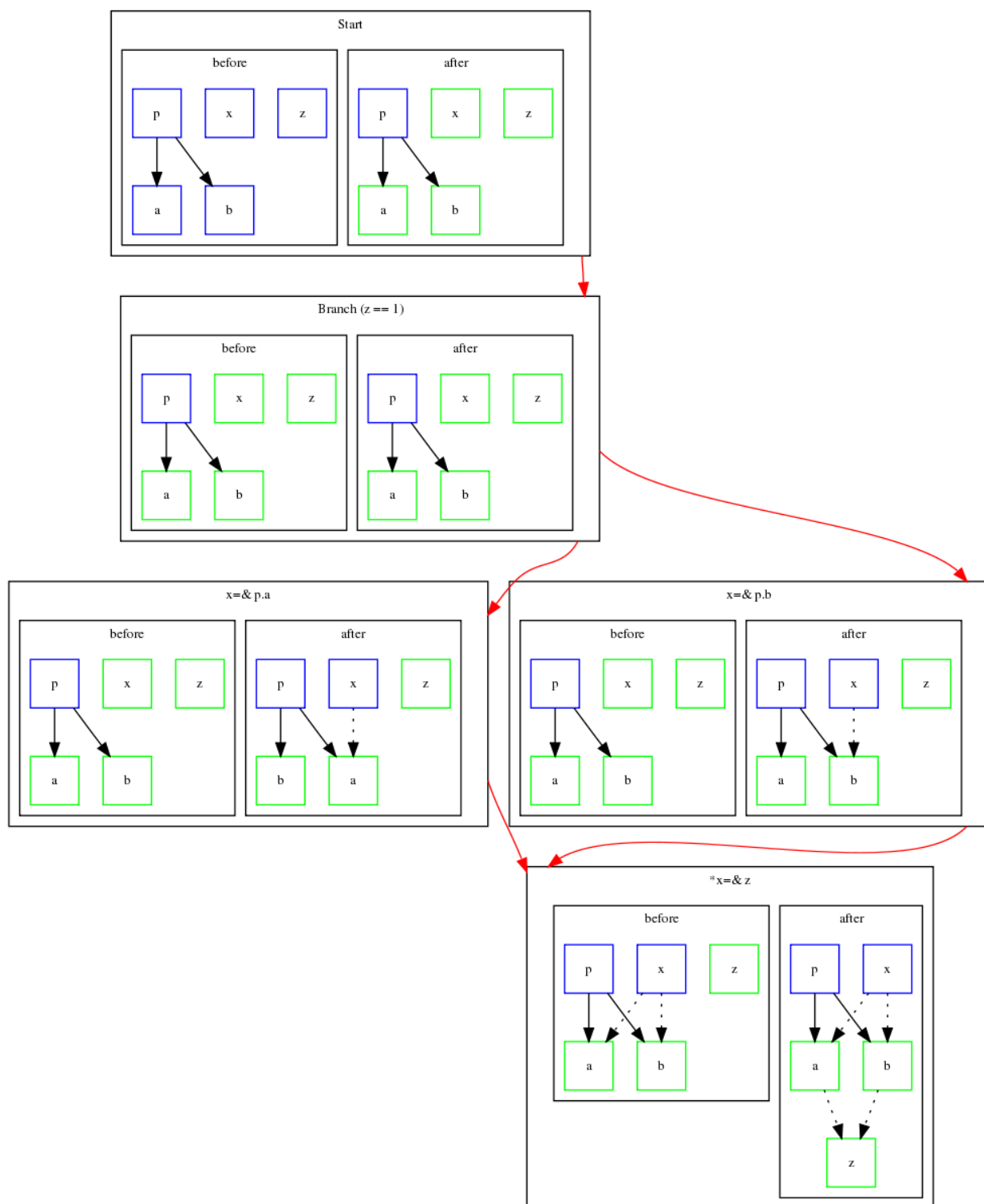


Рис. 6.2: Результат анализа указателей

Глава 7

Заключение

Результатом данной дипломной 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. Представлены описания формального синтаксиса и формальной семантики языка.

Предложенный алгоритм анализа указателей определен в качестве частного случая общего алгоритма анализа потока данных с соответствующей полурешеткой свойств и функциями перехода; доказана корректность алгоритма по отношению к семантике языка Alias.

Произведено встраивание анализа в систему CIL, что дало возможность проверить достаточность выразительности языка Alias 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, для которого существует в свободном распространении средство визуализации Graphviz.

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

Литература

- [1] Horwitz S. *Precise flow-insensitive may-alias analysis is NP-hard*. ACM Transactions on programming languages and systems Vol 19 No 1, Jan. 1997.
- [2] Landi W. *Undecidability of static analysis*. ACM Letters on Programming Languages and Systems Vol 1 No 4 (Dec. 1992), pp. 323 - 337.
- [3] Hind M. *Pointer Analysis: Haven't we solved the problem yet?* ACM Workshop on Program Analysis for Software Tools and Engineering, pp. 54-61, 2001
- [4] Emami M., Rakesh G., Hendren L. J. *Context-sensitive interprocedural points-to analysis in the presense of function pointers*. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 242 - 256, June 1994.
- [5] Chase D. R., Wegman M., Zadeck F. K. *Analysis of Pointers and Structures*. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 296 - 310, 1992.
- [6] Jones N.D., Muchnik S.S. *Program Flow Analysis, Theory and Applications*. Prentice-Hall, 1981.
- [7] Horwitz S., Pfeiffer P. and Reps. T. *Dependence analysis for pointer variables*. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pp. 28-40.
- [8] Landi W., Ryder B. G. *A Safe Approximate Algorythm for Interpocedural Pointer Aliasing*. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 235 - 248, 1992.
- [9] Choi J., Burke M. and P.Carini *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, pp.233-245, January 1993.
- [10] Ghiya R., Hendren L.J. *Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap directed pointers in C.* ACM Symposium on principles of programming languages, 1996.
 - [11] Diwan A., McKinley K. S and Eliot J. *Type-based alias analysis.* ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pp. 106 - 117.
 - [12] Steensgaard B., *Points-to analysis in almost linear time.* ACM Symposium on principles of programming languages, 1996.
 - [13] Andersen L. O. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.
 - [14] 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.
 - [15] Neilson F., Nielson H. R., Hankin C. *Principles of Program Analysis.* Springer, Berlin, 1999.
 - [16] Neilson F., Nielson H. R. *Semantics with Application.* Wiley, 1992.