С.Сейфуллин атындағы Қазақ агротехникалық университетінің **Ғылым жаршысы (пәнаралық)** = **Вестник науки** Казахского агротехнического университета им. С.Сейфуллина **(междисциплинарный)**. - 2021. - №1 (108). - С.180-189

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ ДЛЯ МОДЕЛИРОВАНИЯ ДИНАМИЧЕСКОЙ ПАМЯТИ

А.Ж. Сатыбалдиева, PhD доктор
А.А. Исмаилова, PhD доктор
Е.С. Голенко, PhD докторант
К.К. Кадиркулов, PhD докторант
Казахский агротехнический университет им. С.Сейфуллина,
проспект Жеңіс, 62, г.Нур-Султан Казахстан , 010011
satekbayeva@gmail.com

Аннотация

В данное время с ростом возможностей вычислительных устройств возникло ощущение того, что не надо уделять внимание оптимальности вычислительных процедур: распределению, записи и хранению данных в памяти вычислительных устройств и т.д. В силу такого подхода мы получаем много неудобств при работе с такими устройствами.

Одним из подходов для оптимального распределения и обработки данных в динамической памяти является исчисление алиасов (alias calculus). В данной работе рассмотрены два метода анализа указателей для проведения исчисление алиасов – логика отделимости и исчисление синонимов, и предложен новый подход к решению проблемы для логики отделимости. Данное исследование в области анализа указателей, в частности, исчисление алиасов, было предложено Бертраном Мейером и является новым подходом в исследовании алиасов.

Целью анализа указателей было статически определить значения указателей, которые могут возникнуть во время исполнения программы. Такой анализ предназначен для того, чтобы найти и устранить в программе ошибки, связанные с тем, что разные выражения могут указывать на одну и ту же область памяти. В общем случае эта задача неразрешима, однако было разработано большое количество приближённых алгоритмов, осуществляющих компромисс между скоростью анализа и точностью предлагаемого решения.

Данное исчисление алиасов предложено для более реалистического языка программирования с автоматической и динамической памятью с различными типами для регулярных данных и адресов с разрешимой адресной арифметикой. Все представленные варианты исчисления (в настоящее время) нечувствительны к потоку управления (как исходное исчисление Б. Мейером), но данное исчисление основано на равенствах (в отличие от исходного исчисления).

Ключевые слова: анализ указателей, исчисление синонимов, логика отделимости, динамическая память, верификация, анализ программ, верифицирующий транслятор

Введение в анализ программ

Анализ программ является неотъемлемой частью самых разнообразных манипуляций с программами, таких как трансляция, преобразование, инструментирование и т.п. Полнота и точность могут существенно влиять на качество последующей обработки и, в частности, на эффективность программного обеспечения. Поэтому задача анализа программ, несомненно, остаётся *актуальной* и важность её будет только возрастать по мере увеличения сложности программных систем и повышения уровня требований к ним.

Анализ указателей состоит из решения двух взаимосвязанных подзадач: во-первых, для любого использования указателя определить множество объектов, на которые он может указывать, и во-вторых, для пары указателей определить могут ли они в данной точке программы указывать на один и тот же объект. Вторая подзадача является основной и называется проблемой синонимов. Эти задачи должны решаться и как-то решаются в любом оптимизирующем компиляторе. Очевидно, что существует тривиальная аппроксимация: будет безопасно считать, что любые два указателя могут в любой точке программы являются синонимами. Однако, такое решение запрещает проведение многих оптимизаций.

Средства оптимизации программ нужны для получения высокой скорости исполнения программ или улучшения других характеристик программы. Под оптимизацией программ понимается преобразование программы в семантически эквивалентную, но более эффективную относительно некоторого заданного критерия. Преобразование программы A в программу B эквивалентно (или корректно), если из того, что программа A выполнима на некотором наборе данных, следует, что и B также выполнима на этом наборе и дает тот же результат, что и A. В общем случае задача проверки эквивалентности двух программ неразрешима, и не существует алгоритма, который по данной программе находил бы эквивалентную ей и оптимальную относительно заданного критерия [1].

Тем не менее, существует набор известных оптимизирующих преобразований, таких, что корректность каждого из них гарантирует корректность их последовательного применения. И чтобы конкретное преобразование данной программы было корректным, необходимо выполнение некоторых условий. Например, чтобы иметь возможность убрать генерацию некоторой части программы, нужно быть уверенным, что управление никогда не попадет в эту часть программы. Для получения подобной информации используют

результаты статического анализа. Анализируя код программы, делаются выводы о тех или иных свойствах программы, необходимых для проведения преобразования.

Статический анализ применяется не только для проверки корректности преобразований, но и в инструментах статического анализа кода программ, которые могут находить потенциальные ошибки и определять другие свойства программы без ее непосредственного исполнения.

Существует множество видов статического анализа, и одним из них является анализ указателей и синонимов.

Анализ указателей это один из видов статического анализа, который позволяет определить на какие объекты в памяти могут указывать выражения ссылочного типа в программе, такие объекты называются целями выражения ссылочного типа. Анализ синонимов похож на анализ указателей, его целью является определение, могут ли два разных выражения ссылаться на одно и то же место в памяти (такие выражения называют синонимами).

Существует множество алгоритмов анализа указателей и синонимов, одним из них является алгоритм анализа, основанный на типах (Type-Based Alias Analysis) [2], применимый для языков со строгой типизацией. В языках со строгой типизацией любая ссылочная переменная формального типа T может указывать на любые объекты типа T или его наследников. Простейшая реализация алгоритма основанного на типах дает результат такой, что независимо от контекста и потоков данных в программе целями выражения ссылочного типа являются все объекты совместимые по присваиванию с этим выражением. Такой алгоритм работает быстро, но обладает сравнительно низкой точностью. Анализы нечувствительные к потоку управления вычисляют единственное решение для всей программы или для каждого метода. В то время как анализы чувствительные к потоку управления вычисляют решение для каждого входа программы. Анализы нечувствительные к потоку управления являются либо основанными на равенствах (equality-based) [3], которые обрабатывают присваивание как двунаправленную операцию и обычно используют структуры данных типа объединить-найти, либо основанными на подмножествах (subset-based), которые обрабатывают присваивание как направленный поток значений.

Для сравнения точности алгоритмов анализа указателей нам необходимо ввести некую меру точности. В качестве простой меры точности алгоритма можно использовать усредненное количество синонимов для переменных ссылочного типа, появляющихся в программе [4]. Понятно, что для «идеального» алгоритма анализа это число будет минимальным, а для самого консервативного алгоритма максимальным.

Более точные алгоритмы анализа учитывают не только типы переменных, но и потоки данных в программе. Например, если существует только одно присваивание переменной нового объекта типа T, выделенного в куче, то можно гарантировать, что эта переменная может указывать только на этот объект. С присваиванием значения одной переменной другой переменной ситуация сложнее.

Для дедуктивной верификации объектно-ориентированных программ требуется знать, могут ли два ссылочных выражения в данной точке программы указывать на один и тот же объект во время ее выполнения (иначе говоря, может ли одно выражение быть псевдонимом другого). Например, если операция модифицирует значение атрибута $x.a$, а x и y ссылаются на один и тот же объект, то эта операция изменит и $y.a$, несмотря на то, что при вызове операции и в ее тексте y не упоминается. Исчисление псевдонимов призвано дать ответ на поставленный вопрос[3, с. 77].

Известны следующие подходы к решению проблемы алиасов: shape analysis, separation logic, ownership types и dynamic frames. Однако shape analysis и separation logic пытаются выявить более детальную структуру указателей, чем это необходимо для анализа псевдонимов [3, с. 98]. Логика отделимости, ownership types и dynamic frames требуют от программиста написания дополнительных аннотаций помимо обычных хоаровских утверждений. Исчисление алиасов сформулировано в терминах модельного объектно-ориентированного языка и может потребовать дополнительных аннотаций только в исключительных случаях.

Использование динамического распределения структур данных, т.е., структур, где на обновляемое поле могут сослаться от нескольких точек, широко распространено в таких областях, как системное программирование и искусственный интеллект[5].

Проблема, с которой сталкивается этот подход, является в корректности программы, которая мутирует (изменяет) структуру данных, как правило, зависит от сложных ограничений динамической памяти в этих структурах.

Наш анализ основывается на работах над исчислениями алиасов (синонимов) и логикой отделимости[3, с. 102; 5, с. 58].

В статье рассматривается *анализ указателей* (pointeranalysis), который позволяет судить о корректности программ, а точнее, её правильному обращению с динамической памятью, а также позволяет улучшить её производительность.

Цель данной статьи – дать краткий обзор двум методам анализа указателей – так называемое исчисление синонимов (Calculus of Aliasing) Бертрана Мейера [3, с. 110] и логику отделимости (Separation Logic) [5, с.62], основы которой были заложены Джоном С. Рейнольдсом. Исчисление синонимов - это денотационная семантика языка программирования предназначена для обнаружения программных дефектов или ошибок,

обусловленных тем, что несколько разных выражений указывают на одну область памяти. Логика отделимости - это расширение логики первого порядка разделительной коньюнкцией $*$ и импликацией $-\!\!*$, которые оперируют *непересекающимися* участками динамической памяти; разделительные коньюнкцию и импликацию можно использовать в аксиоматической семантике в аннотациях программ наряду с обычными логическими конструкциями.

Простейшие примеры программ, содержащие ошибки в управлении памятью [6]:

–x = malloc(sizeof(int));

–x = malloc(sizeof(int));//memory leak;

–y = x; free x;free y;// invalid memory access.

Первый пример демонстрирует, как ссылка была утеряна без удаления этого объекта, что влечёт потерю памяти. Второй пример демонстрирует попытку обращения к удалённому объекту, что может привести к аварийной остановке программы.

Хотя ошибки, приведённые в примерах, кажутся очевидными и легко устранимыми, их можно нередко встретить в реальных программах, но эти операторы могут быть разделены тысячами строк и даже могут находиться в разных функциях, что существенно усложняет их поиск. Поэтому создание транслятора осуществляющего поиск и устранение подобных ошибок является одной из главных задач в современном теоретическом программировании (например, исследование верифицирующего компилятора [7]).

Общие определения

Верифицирующий транслятор - это системная программа, переводящая программы, написанные человеком, в эквивалентные низкоуровневые программы, и, кроме того, доказывающая, что обе программы обладают свойствами, специфицированными программистом [8].

Цель анализа указателей – статически определить значения указателей, которые могут возникнуть во время исполнения программы. Такой анализ предназначен для того, чтобы найти и устранить в программе ошибки, связанные с тем, что разные выражения могут указывать на одну и ту же область памяти. В общем случае эта задача неразрешима, однако было разработано большое количество приближённых алгоритмов осуществляющих компромисс между скоростью анализа и точностью предлагаемого решения [4, с. 55]. Время работы различных алгоритмов в худшем случае варьируется от почти линейного до экспоненциального. Рассмотрим текущие направления изучения анализов указателей и открытые задачи, возникшие за долгую историю развития этой области.

Существуют несколько параметров [4, с. 59] влияющих на

соотношение между скоростью и точностью анализов указателей. Некоторые из них:

– чувствительность к потоку управления. Используется ли во время анализа информация о потоке управления? Анализы нечувствительные к потоку управления вычисляют единственное решение для всей программы или для каждого метода. В то время как анализы чувствительные к потоку управления вычисляют решение для каждого входа программы. Анализы нечувствительные к потоку управления являются либо основанными на равенствах (equality-based), которые обрабатывают присваивание как двунаправленную операцию и обычно используют структуры данных типа объединить-найти, либо основанными на подмножествах (subset-based), которые обрабатывают присваивание как направленный поток значений.

– чувствительность к контексту вызова;

– моделирование структур данных;

– требование всей программы;

– представление синонимов.

Даже с выбранной моделью памяти, значение не имеет особого смысла само по себе, т.к. программа может содержать указатели, которые могут указывать на различные объекты во время исполнения.

Несмотря на десятилетия исследования, разработке и использования в этой области, все еще много открытых задач в анализе указателей:

– масштабируемость. Основанные на равенствах и нечувствительные к потоку управления методы анализа способны быстро обрабатывать миллионы строк. Однако, несмотря на то, что недавние исследования улучшили точность таких методов, до сих пор неясно является ли такая точность существенной. В то же время была проделана значительная работа по увеличению скорости более точных основанных на подмножествах методов анализа;

– улучшение точности;

– разработка соответствующих видов анализа для различных классов задач;

– рассмотрение всего потока управления;

– рассмотрение контекста вызываемой функции;

– java и объектно-ориентированные языки. Очевидно, что должны быть созданы новые методы анализа указателей, которые могли бы использовать преимущества новых языков и парадигм программирования (например, строгая типизация указательных переменных в Java);

– рассмотрение отдельных кусков программы (библиотек, функций и т.д.);

– работа анализа указателей в реальных приложениях. На данный момент недостаточно хорошо исследовано то, как методы анализа работают на «реальных» приложениях, которые могут содержать большие программы, быть многопоточными, использовать

библиотеки и т.д., то есть в условиях далёких от идеальных.

В силу этих и других причин появилось новое исследование в области анализа указателей. В частности, исчисление алиасов, предложенное Бертраном Мейером, является новым подходом к исследовании алиасов. Три варианта исчисления алиасов для игрушечных императивных языков с единственным типом данных для абстрактных указателей представлены в [9]; это исчисление, основанное на множестве формализмов нечувствительных к потоку управления и контекста, и без адресной арифметики.

Данное исчисление алиасов предложено для более реалистического языка программирования с автоматической и динамической памятью с различными типами для регулярных данных и адресов с разрешимой адресной арифметикой. Все представленные варианты исчисления (в настоящее время) нечувствительны к потоку управления (как исходное исчисление Б. Мейером), но данное исчисление основано на равенствах (в отличие от исходного исчисления).

В статье [10] представлен язык программирования MoRe, который является диалектом языка программирования, используемого для определения логики отделимости (SeparationLogic) в [5, с.58], MoRe сокращенно от *MoreRealistic, т.е.* более реалистичный язык.Язык имеет два типа данных, которые называются адреса и целые числа с неявным преобразованием типа от целых чисел до адресов.

Исчисление алиасов, основанных на стэке, для языка программирования MoRe

Для объединения подходов - исчисление алиасов и логики отделимости - предлагается определить отношение синонимичности Sна множестве всех адресных выражений встречающихся в программе E (замкнутое относительно подвыражений). Адресное выражение здесь - это любое выражение, построенное из натуральных чисел и переменных при помощи сложения и вычитания, которое используется в операторах выделения памяти, косвенного присваивания, разыменования или освобождения памяти. Для любой адресной переменной x из E обозначим через $E{\sim}x$ множество тех выражении из E, которые используют переменную x, а через $E\backslash x$ множество тех выражений из E, которые не используют переменную x; очевидно, что $E = (E{\sim}x) \cup (E\backslash x)$. Пара синонимов – это произвольное равенство адресных выражений (т.е. диофантово уравнение первой степени) из E. Пара антонимов – это произвольное неравенство ($\neq$) адресных выражений из E (т.е. диофантово неравенство первой степени) [11, с. 512].

Зафиксируем программу. Множество адресных переменных AV и адресных выражений AE (программы) определяются

совместной индукцией следующим образом:

а) адресными переменными являются любая переменная x которая появляется (в программе):

– в левой стороне любого выделения памяти $x := cons \ldots$,

– в левой стороне любого косвенного присваивания $[x] := \cdots$,

– в правой стороне любого разыменования $\ldots := [x]$,

– в любом операторе освобождения памяти $dispose(x)$,

– в любом адресном выражений;

б) адресными выражениями являются (в программе):

– все адресные переменные;

– все подвыражения любых адресных выражений;

– все выражения τ, построенные из C и V при помощи сложения и вычитания, которые появляются в правой стороне любого присваивания в любой адресной переменной $x := \tau$;

– все выражения $x + 1, \ldots x + k$ такие, что программа имеет выделение памяти $x{:=}cons\,\tau_0 \ldots \tau_k$.

Пусть для любого множества адресных выражений AS и любого множества адресных переменных x из AV:

– $AS{\sim}x$ - множество тех выражении в AS, вкоторых используется x(т.е. где появляется x);

– $AS\backslash x$ - множество тех выражении в ASвкоторых не используется x(т.е. где не появляется x);

Очевидно, что $AS = (AS\backslash x) \cup (AS{\sim}x)$. Для любого множества адресных выражений AS любого множества адресных переменных $D \subseteq AV$ оставим *AS(D)* как множество всех адресных выражений в *AS* которые не используют переменные кроме *D* (i.e. $AS(D) = \cap_{x \notin D} AS\backslash x$).

Парой алиасов (*синонимы*) является равенство двух адресных выражений. Парой антиалиасов (*антонимы*) является неравенство ($\neq$) из двух адресных выражений.

Напомним, что все адресные выражения в AE являются линейными выражениями с целыми коэффициентами. Это означает, что пары синонимов или антонимов в AE выражаются в виде диофантовых уравнений и неравенств над целыми числами. Но мы думаем, что все эти пары как равенства и неравенства над $(ADR, 0, 1, +, -)$ при условии, неявных приведенных типов (применяется ко всем используемых целочисленных констант).

Конфигурация – это произвольная конечная совокупность троек *Cnf=(I, N, S)* состоящая из двух множеств $N \subseteq I \subseteq AV$ адресных переменных и конечного множества *S*пар синонимов и антонимов (с переменными в *I*), которая имеет решение как система диофантовых равенств и неравенств (в арифметике Пресбургера или кольце вычетов) в (*ADR, 0, 1, +, –*), т.е. которая

согласуется с теорией T_{ADR}, неформально множество I является адресным переменным для инициализации, множество N является адресным переменным для нераспределенной инициализации, а множество S является системой равенств и неравенств, чтобы четко определить, какие выражения могут быть алиасами, а какие не могут быть.

Пусть C – произвольная конфигурация, тогда логическая семантика конфигурации C – это конъюнкция $\&C$ всех пар синонимов и антонимов конфигурации; формальная семантика конфигурации C – это теория $[C]$, задаваемая этой конфигурацией (т.е. все пары всех равенств и неравенств, которые следуют из конфигурации); дополнение для C – это совокупность всех пар антонимов, непротиворечащих теории $[C] comp(C) = \{e' \neq e'' : \{e' = e''\} \notin [C]\}$;

замыкание C – это конфигурация $close(C) = C \cup comp(C)$. Пусть st – произвольное состояние стека; будем писать $st \vDash C$и говорить, что stудовлетворяет конфигурации C, когда в st выполнены все формулы из $close(C)$ [12, 13].

Пусть для любой конфигурации *Cnf=(I, N, S)*:

– *&Cnf* - конъюнкция всех пар синонимов и антонимов в *S*(при условии, неявное приведение типа);

– $cls(Cnf) = \{e' = e'' : e', e'' \in AE(I), T_{ADR} \vdash \&Cnf \rightarrow (e' = e'')\} \cup \{e' \neq e'' : e', e'' \in AE(I), T_{ADR} \vdash \&Cnf \rightarrow (e' \neq e'')\}$;

– $ncl(Cnf) = cls(Cnf) \cup \{e' \neq e'' : e', e'' \in AE(I), (e' = e'') \notin cls(Cnf)\}$.

Пусть *st*- состояние стэка, мы пишем $st \vDash Cnf$ и говорим, что *st*удовлетворяет конфигурации C, переменные из *I*выполнены в *st*(т.е. $I \subseteq dom(st)$) и все формулы $ncl(Cnf)$ являются истинными в *st*(i.e. $st \vDash \&ncl(Cnf)$).

Для любой конфигурации $Cnf' = (I', N', S')$ и $Cnf'' = (I'', N'', S'')$ скажем что, они эквивалентны, если $I' = I''$, $N' = N''$ и$ncl(Cnf') = ncl(Cnf'')$.

Распределение (или распределение алиасов) - это произвольная конечная совокупность конфигураций, в которой ни одна не поглощается (т.е. не является логическим следствием) другой. Если D – произвольная совокупность конфигураций, то очистка D – это распределение алиасов $rfn(D)$, получающееся изDв результате удаления всех конфигураций, которые поглощаются другими конфигурациями.

Пусть D – произвольное распределение алиасов, а st – произвольное состояние стека; будем писать $st \vDash D$и говорить, что stудовлетворяет распределению D, когда $st \vDash Cnf$ хотя бы для одной конфигурации *Cnf* из D.

Определим преобразователь aft распределений для каждой программы языка MoRe логики отделимости индукцией по структуре программ: база индукции определяется преобразователем для отдельных операторов; шаг индукции определяется преобразователем для составных программ.

Индивидуальные операторы

Для операторов, не изменяющих адресные переменные, имеем:

- $aft(D, skip) = D$;
- $aft(D,\ varx := i) = D$, если x не является адресной переменной;
- $aft(D, x := t) = D$, если x не является адресной переменной;
- $aft(D, x := [t]) = D$, если x не является адресной переменной;
- $aft(D, [x] := t) = D$, если t не является адресным выражением.

Если x является адресной переменной, то распределение $aft(D, var\ x = i)$ получается следующим образом. Пусть $Cnf = (I, N, S)$ – произвольная конфигурация из D. Проведем реинициализацию, если $x \in I$. Пусть $Cnf_{varx=i} = (I_{varx=i}, N_{varx=i}, S_{varx=i})$, где:

- $I_{var\ x=i} = I \cup \{x\}$,
- $N_{var\ x=i} = N$, и $S_{var\ x=i} = ncl\{e' = e'' : e', e'' \in AE(I_{var\ x=i})$ и $\boldsymbol{T_{ADR}} \vdash \&Cnf \rightarrow (e'_{i/x} = e''_{i/x})\}$
- _

.

Тогда $aft(D,\ varx = i)$ - это $rfn\{Cnf_{varx=i} : Cnf \in D\}$.

Если x - некоторая адресная переменная, то распределение $aft(D, x := t)$ получается следующим образом. Пусть $Cnf = (I, N, S)$ – произвольная конфигурация из D. Не инициализируется, если $x \notin I$ или t будет неинициализируемой переменной (т.е. не в I). Пусть $Cnf_{x:=t} = (I_{x:=t}, N_{x:=t}, S_{x:=t})$ где:

- $I_{x:=t} = I$;
- $N_{x:=t} = N$;
- $S_{x:=t} = ncl\{e' = e'' : e', e'' \in AE(I)$ и $\boldsymbol{T_{ADR}} \vdash \&Cnf \rightarrow (e'_{t/x} = e''_{t/x})\}$
- _

.

Тогда $aft(D,\ x := t)$ - это $rfn\{Cnf_{x:=t} : Cnf \in D\}$.

Распределение $aft(D, x := cons(t_0, \ldots t_k))$ получается следующим образом. Пусть $Cnf = (I, N, S)$ – произвольная конфигурация из D. Не инициализируется, если $x \notin I$ или любое tв $t_0, \ldots t_k$ является неинициализируемой переменной (т.е. не в I). Утечка памяти, если $\boldsymbol{T_{ADR}} \nvdash \&Cnf \rightarrow (e = x)$ для каждого аврескного выражения $e \in AE(I) \backslash x$. Пусть y – новая переменная и пусть $New_{Cnf}\ (y, k)$ – множество всех пар антонимов вида $e \neq y + i$, или $y + i \neq y + j$, где $e \in AE(I)$ и $0 \leq i < j \leq k$. Будет вне памяти, если S несовместим с $New_{Cnf}\ (y, k)$. Пусть $Cnf_{x:=cons(t_0,\ldots t_k)}$ - $(I_{x:=cons(t_0,\ldots t_k)}, N_{x:=cons(t_0,\ldots t_k)}, S_{x:=cons(t_0,\ldots t_k}$ где:

– $I_{x:=cons(t_0,\dots t_k)} = I$;

– $N_{x:=cons(t_0,\dots t_k)} = N \backslash \{x\}$;

– $S_{x:=cons(t_0,\dots t_k)} == ncl\{e' = e'' : e', e'' \in AE(I), \boldsymbol{T_{ADR}} \vdash (\&Cnf\ \&New_{Cnf}(y,k)) \rightarrow (e'_{y/x} = e''_{y/x})\}$

.

Тогда $aft(D, x := cons(t_0, \dots t_k))$ - это $rfn\{Cnf_{x:=cons(t_0,\dots t_k)} : Cnf \in D\}$.

Если x является адресной переменной, то распределение $aft(D, x := [t])$ получается следующим образом. Пусть $Cnf = (I, N, S)$ – произвольная конфигурация из *D*. Не инициализируется, если $x \notin I$ или *t*является неинициализируемой переменной (т.е. не в *I*). Пусть $Cnf_{x:=[t]}$ это множество конфигурации $(I_{x:=[t]}, N_{x:=[t]}, S')$ где:

– $I_{x:=[t]} = I$;

– $N_{x:=[t]} = N$;

– S' совместима с $ncl\{e' = e'' : e', e'' \in AE(I \backslash x)$ и $\boldsymbol{T_{ADR}} \vdash (\&Cnf \rightarrow (e' = e''))\}$

.

Тогда $aft(D,\ x := [t])$ - это $rfn(\cup_{Cnf \in D}\ Cnf_{x:=t})$.

Распределение $aft(D, dispose(t))$получается следующим образом. Пусть $Cnf = (I, N, S)$ – произвольная конфигурация из *D*. Не инициализируется, если $x \notin I$.не выделяется память, если $x \in N$. $Cnf_{dispose(x)} = (I_{dispose(x)}, N_{dispose(x)}, S_{dispose(x)})$

где:

– $I_{dispose(x)} = I$;

– $N_{dispose(x)} = N \cup \{x\}$;

– $S_{dispose(x)} = S$.

Тогда $aft(D,\ dispose(x))$ - это $rfn\{Cnf_{dispose(x)} : Cnf \in D\}$.

Составные программы:

– $aft(D, (\alpha; \beta)) = aft(\ aft(D, \alpha), \beta)$;

– $aft(D, if\ \varphi\ then\ \alpha\ else\ \beta) = rfn(aft(D, \alpha) \cup aft(D, \beta))$;

– $aft(D, while \varphi do \alpha) = rfn(\cup_{i \geq 0} aft(D, \alpha^i))$,

где $\alpha^0 \equiv skip$, и $\alpha^{i+1} \equiv \alpha^i$; для любого $i \geq 0$.

Утверждение. Для любой *MoRe*-программы α языка программирования для логики отделимости, для любого распределения алиасов D утверждение частичной корректности $\{D\}\alpha\{aft(D, \alpha)\}$является истинным, т.е. для любых состояний стека st и st', если $st \vDash D$ и программа α, начав работу в состоянии $(st, \_)$, завершает работу в состоянии $(st', \_)$, то $st' \vDash aft(D, \alpha)$[12, с. 533].

Доказательство: индукция по длине программы.

База индукции: В случаях операторов, не изменяющих значения адресных выражений, когда утверждение очевидно.

Пусть $\alpha = \{x := t\}$, и x является адресной переменной, D – произвольное распределение

синонимов, st - состояние стека такое, что $st \vDash D$, $st' = upd(st, x, st(t))$, $D' = aft(D, x := t)$. Надо доказать, что $st' \vDash D'$.Предположим, что D состоит из одной конфигурации C, пусть $C' = C_{x:=t}$. Докажем, что $st' \vDash C'$.

$C' = \{e' = e'' : e', e'' \in E;\ C \vdash e'_{t/x} = e''_{t/x}\} \cup \{e' \neq e'' : e', e'' \in E;\ C \vdash e'_{t/x} \neq e''_{t/x}\}$

.

Пусть $\{e' = e''\} \in C'$, тогда $C \vdash e'_{t/x} = e''_{t/x}$, значит $st \vDash e'_{t/x} = e''_{t/x}$. Но так как $st' = upd(st, x, st(t))$, то $st' \vDash e'_{t/x} = e''_{t/x}$и, $st' \vDash e' = e''$ как следствие $st' \vDash t = x$. Аналогично если $\{e' \neq e''\} \in C'$, то $st' \vDash e' \neq e''$.

Выводы

Данная статья посвящена*анализу совпадения указателей* (*aliasing*), что призвано помочь в доказательстве корректности программ с динамической памятью, рассмотрены два метода анализа указателей – логика отделимости и исчисление синонимов, и предложен новый подход к решению проблемы для логики отделимости.

Представлен язык программирования MoRe, который является диалектом языка программирования, используемого для определения логики отделимости.

Целью анализа указателей было статически определить значения указателей, которые могут возникнуть во время исполнения программы. Такой анализ предназначен для того, чтобы найти и устранить в программе ошибки, связанные с тем, что разные выражения могут указывать на одну и ту же область памяти. В общем случае эта задача неразрешима, однако было разработано большое количество приближённых алгоритмов осуществляющих компромисс между скоростью анализа и точностью предлагаемого решения. Время работы различных алгоритмов в худшем случае варьируется от почти линейного до экспоненциального. Рассмотрено текущие направления изучения анализов указателей и открытые задачи, возникшие за долгую историю развития этой области.

Данное исчисление алиасов предложено для более реалистического языка программирования с автоматической и динамической памятью с различными типами для регулярных данных и адресов с разрешимой адресной арифметикой. Все представленные варианты исчисления (в настоящее время) нечувствительны к потоку управления (как исходное исчисление Б. Мейером), но данное исчисление основано на равенствах (в отличие от исходного исчисления).

Для объединения этих двух подходов предложено определить отношение синонимичности S на множестве всех адресных выражений встречающихся в программе E (замкнутое относительно подвыражений).

Доказано утверждение о частичной корректности $\{D\}\alpha\{aft(D,\alpha)\}$Для любой *MoRe*-программы α языка программирования для логики отделимости, для любого распределения алиасов.

Список литературы

1 Касьянов В.Н. Методы построения трансляторов. – Новосибирск: Наука, 1986. – 344 с.

2 Diwan A. Type-based alias analysis // Proceedings of the ACM SIGPLAN conference on Programming language design and implementation. – New York, 1998. – P. 106–117.

3 Meyer B. Steps Towards a Theory and Calculus of Aliasing // International Journal of Software and Informatics. – 2011. – Vol. 5, №1-2. – P. 77-115.

4 Hind M. Pointer Analysis: Haven't We Solved This Problem Yet? // Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. –New York, 2001. – P. 54-61.

5 Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // Proceedings of 17^{th}IEEE Symposium on Logic in Computer Science. – Copenhagen, 2002. – P. 55-74.

6 Haberland R., Ivanovskiy S. Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog // Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering. – Saint Pitersburg, 2014. – P. 17-23.

7 Andersen L.O. Program Analysis and Specialization for the C Programming Language: PhD thesis. – DIKU: University of Copenhagen, 1996. – 157 p.

8 Steensgard B. Points-to analysis in almost linear time // Conference Record of the 24th ACM SIGPLAN – SIGACT Symposium of Principles of Programming Language. – London, 1997. – P. 32-41.

9 Hoare C.A.R. The Verifying Compiler: A Grand Challenge for Computing Research // Perspectives of Systems Informatics. – 2003. –№2890. – P. 1-12.

10Сатекбаева А.Ж., Тусупов Д.А. Два формализма для указателей: логика отделимости и исчисление алиасов // Вестник ВКГТУ. – 2013. – Ч. 1. – С. 31-37.

11 Воронцов А., Сатекбаева А.Ж., Шилов Н.В. Исчисление алиасов (синонимов) для логики отделимости // Интеллектуальные информационные и коммуникационные технологии – средство осуществления третьей индустриальной революции в свете Стратегии «Казахстан-2050»: матер. междунар. конф. – Астана, 2014. – С. 512-515.

12 Воронцов А., Сатекбаева А.Ж., Тусупов Д.А. и др. Исчисление алиасов (синонимов) для простого императивного языка с адресной

арифметикой // Теоретические и прикладные проблемы математики, механики и информатики: матер. междунар. науч. конф. – Караганды, 2014. – С. 77-78.

13 Сатекбаева А., Тусупов Д. Сравнение двух формализмов для указателей // Интеллектуальные информационные и коммуникационные технологии - средство осуществления третьей индустриальной революции в свете стратегии «Казахстан-2050»: матер. междунар. конф. – Астана, 2013. – С. 533-535.

References

1 Kasianov V.N. Metody postroeniia transliatorov. – Novosibirsk: Naýka, 1986. – 344 s.

2 Diwan A. Type-based alias analysis // Proceedings of the ACM SIGPLAN conference on Programming language design and implementation. – New York, 1998. – P. 106–117.

3 Meyer B. Steps Towards a Theory and Calculus of Aliasing // International Journal of Software and Informatics. – 2011. – Vol. 5, №1-2. – P. 77-115.

4 Hind M. Pointer Analysis: Haven't We Solved This Problem Yet? // Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. –New York, 2001. – P. 54-61.

5 Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // Proceedings of 17th IEEE Symposium on Logic in Computer Science. – Copenhagen, 2002. – P. 55-74.

6 Haberland R., Ivanovskiy S. Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog // Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering. – Saint Pitersburg, 2014. – P. 17-23.

7 Andersen L.O. Program Analysis and Specialization for the C Programming Language: PhD thesis. – DIKU: University of Copenhagen, 1996. – 157 p.

8 Steensgard B. Points-to analysis in almost linear time // Conference Record of the 24th ACM SIGPLAN – SIGACT Symposium of Principles of Programming Language. – London, 1997. – P. 32-41.

9 Hoare C.A.R. The Verifying Compiler: A Grand Challenge for Computing Research // Perspectives of Systems Informatics. – 2003. –№2890. – P. 1-12.

10 Satekbayeva A.J., Tussupov J.A. Dva formalizma dlia ukazatelei: logika otdelimostiiischislenie aliasov // Vestnik VKGTU. – 2013. – Ch. 1. – S. 31-37.

11 Vorontsov A., Satekbayeva A.J., Shilov N.V. Ischislenie aliasov (sinonimov) dlia logiki otdelimosti // Intellektualnye informatsionnye i kommunikatsionnye tehnologii – sredstvo osýestvleniia treteiindýstrialnoi revoliutsii v svete Strategii «Kazahstan-2050»: mater. mejdunar. konf. – Astana, 2014. – S. 512-515.

12 Vorontsov A., Satekbayeva A.J., Tussupov J.A. i dr. Ischislenie aliasov (sinonimov) dlia prostogo imperativnogo iazyka s adresnoi arifmetikoi // Teoreticheskie i prikladnye problemy matematiki, mehanikiiinformatiki: mater. mejdunar. naých. konf. – Karagandy, 2014. – C. 77-78.

13 Satekbayeva A.J., Tussupov J.A. Sravnenie dvuh formalizmov dlia ukazatelei // Intellektualnye informatsionnye i kommunikatsionnye tehnologii - sredstvo osuestvleniia treteiindustrialnoi revoliutsii v svete strategii «Kazahstan-2050»: mater. mejdunar. konf. – Astana, 2013. – S. 533-535.

ДИНАМИКАЛЫҚ ЖАДЫНЫМОДЕЛЬДЕУГЕ АРНАЛҒАН ДЕРЕКТЕРДІҢ АБСТРАКТІ ТИПІ

А. Ж. Сатыбалдиева, PhD доктор
А.А. Исмаилова., PhD доктор
Е.С. Голенко, PhD докторант
Қ.Қ. Қадірқұлов, PhD докторант
С.Сейфуллин атындағы Қазақ агротехникалық университеті,
Жеңіс даңғылы, 62, Нұр-Сұлтан қ., 010011, Қазақстан
satekbayeva@gmail.com

Түйін

Мақалада деректердің жаңа абстрактілі типтері мен көрсеткіштерін енгізу арқылы білімді бейнелеудің формальды-тұжырымдамалық тәсілдері сипатталған, сонымен қатар осы саланың дамуының ұзақ тарихында туындаған көрсеткіштік талдаулар мен ашық мәселелерді зерттеудің қазіргі бағыттары талқыланады.

Бағдарламалардың тиімділігін, дәлірек айтсақ, оның динамикалық жадымен дұрыс жұмыс істеуін бағалауға мүмкіндік беретін, сонымен қатар оның жұмысын жақсартатын көрсеткіштерге салыстырмалы талдау (pointer analysis) жүргізілді.

Көрсеткіштерді талдаудың екі әдісі: алиастарды есептеу (Calculus of Aliasing) және бөлу логикасын есептеу (Separation Logic)туралы қысқаша сипаттама берілді, MoRe бағдарламалау тілі, оның формальды синтаксисі және құрылымдық оперативті семантикасы жасалған.

Бөлу логикасы, кез-келген алиастарды есептеу үшін көрсеткіштерді талдау арқылы бағдарламаларды синтаксистік тексеруге арналған алгоритмдердің тиімділігінің дәлелдемесі келтірілген.

Кілт сөздер: көрсеткішті талдау, алиастарды есептеу, бөлу логикасы, динамикалық жады, верификация, бағдарламалық талдау, верификациялық транслятор

ABSTRACT DATA TYPES FOR DYNAMIC MEMORY SIMULATION

A.J. Satybaldiyeva PhD
A.A. Ismailova, PhD
E.S. Golenko, PhD doctoral student
K.K Kadirkulov, PhD doctoral student
S.Seifullin Kazakh Agro Technical University
Zhenis avenue, 62, Nur-Sultan, 010011,Kazakhstan

satekbayeva@gmail.com

Abstract

The article describes formal-conceptual approaches for representing knowledge by introducing new abstract data types and pointers, and also discusses the current directions of studying pointer analyzes and open problems that have arisen over the long history of the development of this field.

A comparative analysis of pointers (pointer analysis) has been carried out, which allows judging the correctness of programs, or rather, its correct handling of dynamic memory, and also improves its performance.

A brief description of two methods for analyzing pointers - Calculus of Aliasing and Separation Logic - is given, the subsequent development of the alias calculus is presented, and the MoRe programming language, its formal syntax and structural operational semantics are developed.

The proof of the correctness of algorithms for syntactic checking of programs with analysis of pointers in solving the problem for the logic of separation, for any distribution of aliases is given.

Keywords: pointer analysis, alias calculus, separation logic, dynamic memory, verification, program analysis, verifying translator