Компьютерные инструменты в образовании, 2019
№ 2: 5–??
УДК: 004.052.2
http://cte.eltech.ru
doi:10.32603/2071-2340-2019-2-5-30

СРАВНИТЕЛЬНЫЙ АНАЛИЗ СТАТИЧЕСКИХ МЕТОДОВ ВЕРИФИКАЦИИ ДИНАМИЧЕСКОЙ ПАМЯТИ

Хаберланд Р.[1], аспирант, haberland1@mail.ru

[1]Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина), ул. Профессора Попова, д. 5, корп. 2, 197376, Санкт-Петербург, Россия

Аннотация

В статье дается обзор существующих методов верификации динамической памяти; проводится их сравнительный анализ; оценивается применимость для решения задач управления, контроля и верификации динамической памяти. Данная статья состоит из восьми разделов. Первый раздел — введение. Во втором обсуждаются проблемы управления динамической памятью. В третьем рассматривается исчисление Хоара. В четвёртом речь идёт о преобразованиях heap в стек. Пятый вводит понятие анализа образов динамической памяти. Шестой посвящен ротации указателей, седьмой — логике распределенной памяти. В последнем разделе рассматриваются возможные направления дальнейших научных исследований в данной области, в частности: распознавание на уровне записи различных экземпляров объектов; автоматизация доказательств; использование «горячего» кода, то есть программного кода, который сам себя обновляет при запуске программы; расширение интуитивности объяснений доказательств и другие.

Ключевые слова: *верификация динамической памяти, исчисление Хоара, логика распределенной памяти, операции с указателями.*

Цитирование: Хаберланд Р. Сравнительный анализ статических методов верификации динамической памяти // Компьютерные инструменты в образовании. 2019. № 2. С. 5–30. doi: 10.32603/2071-2340-2019-2-5-30

1. ВВЕДЕНИЕ

Спустя несколько десятилетий, ошибки основанные на неверном использовании динамической памяти остаются одними из самых дорогостоящих ошибок при разработке программного обеспечения. Локализация ошибок является трудной задачей, потому что довольно часто диагностируемое место возникновения ошибки и место настоящей ошибки далеко отстоят друг от друга. Ошибки влияют на общую устойчивость приложения, быстродействие и корректность программы.

В статье дается обзор существующих методов верификация динамической памяти; проводится их сравнительный анализ; оценивается применимость для решения задач управления, контроля и верификации динамической памяти.Перед тем как перейти к моделям верификации динамической памяти, рассмотрим более детально предмет нашего исследования.

стек (stack)
куча (heap)
неинициализированные данные (bss)
инициализированные данные (data)
программный код (text)

Рис. 1. Регионы памяти

Регион динамической памяти выделяется операционной системой при запуске процесса [1]. Из этого региона память выдается приложению по запросу (например с помощью операций malloc или new), на рис. 1 этот регион памяти обозначен как «куча» (*англ.* heap).

В целях избежания путаницы, регион памяти процесса будет далее обозначаться как «*динамическая память*»; а логическая структура данных (граф), определенный в этой непрерывной логической части памяти процесса и содержащий блоки динамической памяти с их описанием, будем именовать как «*heap*».

Программный код содержит процессорные инструкции, то есть операторы вместе с операндами. Инициализированные данные содержат переменные с начальным значением. Неинициализированные переменные являются глобальными и нелокальными переменными, которым не присвоено начальное значение. Между регионами стека и heap четкой границы нет, она плавающая. Стек (точнее, указатель на границу стека) увеличивается на размер стекового кадра при вызове процедуры и уменьшается при выходе. Локальные переменные помещаются в текущий стековый кадр и автоматически освобождаются при выходе. Таким образом, локальные переменные автоматически выделяются и утилизируются в ходе выполнения программы. В отличие от стека heap растёт при явных командах выделения динамической памяти и остаётся аллоцированным до тех пор, пока выделенная память также явно не будет освобождена. В 32-разрядных системах, использующих виртуальную память, активно используется сегментация. Это означает, что доступ к ячейкам памяти, которая организована по сегментам, проводится с помощью управляемых устройств операционной системы. В 64-разрядных операционных системах разделители сегментов процесса всё ещё присутствуют, но практически не используются, так как необходимость сегментации при достаточно большом адресном пространстве отпадает.

Под верификацией подразумевается формальная система проверок, предложенная Хоаром [2], которая используется в алгебраических и логических формулах и логических правилах для доказательства (не-)верности соблюдения некоторой заданной спецификации (также говорят «*исчисление Хоара*»).

В качестве доказательства необходимости верификации динамической памяти рассмотрим следующий фрагмент кода на языке Си, осуществляющий инвертирование связанного списка:

```
struct list_elem{
  int data;
  list_elem *next;
};
```

Пусть инвертирование реализуется следующей процедурой:

```
list_elem *temp;   list_elem *y = NULL;
while (x != NULL) {
  temp = x->next;
  x->next = y;
  y = x;
  x = temp;
}
```

Здесь после каждой итерации цикла динамическая память и указатели x и y выглядят так:

Итерация №1:

y: ⟶ 0   x: ⟶ [1] ⟶ [2] ⟶ [3] ⟶ 0

Итерация №2:

y: ⟶ [1] ⟶ 0   x: ⟶ [2] ⟶ [3] ⟶ 0

Итерация №3:

y: ⟶ [2] ⟶ [1] ⟶ 0   x: ⟶ [3] ⟶ 0

Итерация №4:

y: ⟶ [3] ⟶ [2] ⟶ [1] ⟶ 0   x: ⟶ 0

Нетрудно заметить, что в начале x содержит исходный список, а y пуст. При окончании цикла y содержит исходный список в обратном порядке, а x пуст. Обратим внимание на то, что семантика, даже данная относительно простой программы, не очевидна. Задачей спецификации программы является создание описания состояния динамической памяти до начала исполнения и после его окончания.

2. ПРОБЛЕМЫ УПРАВЛЕНИЯ ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Основным различием между стеком и heap является способ их организации. Стек автоматически (неявно) выделяется и освобождается данной программой при вызове процедур и выходе из них. Управление heap осуществляется явно при помощи специальных операторов. Содержимое памяти меняется постоянно и отражает текущее состояние исполнения. Heap можно представить ориентированным графом, где вершинами являются структуры, хранящие пользовательские данные; а дуги соответствуют полям-указателям. Пример структуры heap приведен на рис. 2. Пример организации стека приведен на рис. 3.

Динамическая память строится пошагово. Размер вершин определяется пользовательским типом. Пошаговое построение подразумевает, что изменения могут быть не замечены ссылающимися указателями. Вершины выделяются в Си с помощью оператора `malloc`, а с помощью `free` производится утилизация свободной памяти процедурами операционной системы [3]. Доступ к вершинам осуществляется по имеющимся в стеке и динамической памяти указателями. Доступ описывается выражением доступа. Недопустимые элементы являются мусором и подлежат утилизации. Указатели, ссылающиеся на одну и ту же ячейку в heap, являются «псевдонимами».

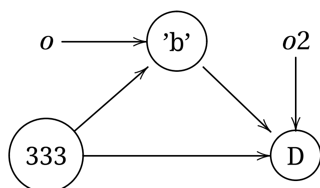

Рис. 2. Структура heap

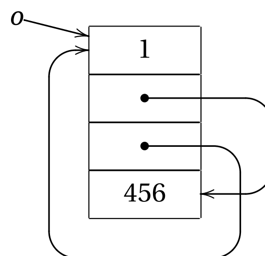

Рис. 3. Организация стека

Нетрудно представить, что работа с динамической памятью может привести к множеству проблем в приложениях, впервые классифицированных Миллером в [4]. Многие из них остаются актуальными по сей день. Одна из ключевых статей по анализу найденных ошибок [5] называет проблемы с указателями и динамической памятью самыми трудновыявляемыми проблемами среди ошибок, обнаруженных при программировании и проектировании. В ней дается сравнение инструментов разработки программного обеспечения под Linux и Windows. Было выявлено, что Linux компоненты содержат значительно меньше ошибок, чем Windows. Это можно объяснить активным участием сообщества в выявлении и исправлении ошибок и большом переиспользовании исходного кода. Наиболее частыми ошибками Миллер называет проблемы с неверными адресами в случае использования динамической памяти, а также с некорректными пределами динамически выделенных массивов. Проблемы трудно выявить, так как место проявления ошибки часто отдалено от ее реальной причины, которая заключается в неверном использовании указателя. Если, например, неверный доступ в сегменте данных приводит к немедленной приостановке работы процесса, то локализация неверной операции довольно легка. В случае динамической памяти поиск некорректной операции над указателем является довольно сложным. Одним из подходов, нацеленных на устранение сложности управления динамической памятью, является преобразование heap в стек, рассматриваемое в разделе 4.

Проблемы использования динамической памяти можно группировать по различным признакам. Мы предлагаем следующую классификацию:

(i) утечка памяти;
(ii) недоступная память;
(iii) неверный доступ к памяти;
(iv) отклонения полученной от ожидаемой структуры данных;
(v) проблемы с псевдонимами и «далёкая манипуляция».

В следующих подразделах приводится подробное описание приведенных проблем с динамической памятью.

2.1. Утечка памяти

Рассмотрим следующий пример:

```
MyClass object1=new MyClass();
...
object1=new MyClass();
```

Предположим, что между первым и вторым присваиванием `object1` не имеется указателей, которые ссылались бы на первый выделенный регион памяти. В таком случае

первый экземпляр класса `MyClass` потерян, то есть является мусором. Если объект не нужен, то лучше его утилизировать, иначе может возникнуть ситуация, когда выделение памяти операционной системой завершается провалом, либо совершается программа с ошибкой, либо программа затормаживается из-за поиска свободного региона памяти в связи с фрагментацией памяти. Сборка мусора может привести к неопределяемому выходу из строя и к неожиданному замедлению работы из-за поиска свободных ресурсов операционной системы. Как правило, на практике поиск и отладка такого рода ошибок сложны и в общем случае не детерминированы. Определение мусора по Вайлю [3] дается с помощью количества ссылающихся указателей, которое равно нулю.

Джоунс и соавторы в [6] представляют фактически полный обзор на тему сборки мусора и детально обсуждают актуальные подходы с акцентом на многопоточные приложения. Далее даются оценки наиболее важным результатам в этой области.

Аппель [7] опасается, что на практике из-за архитектуры фон-Неймана имеются ограничения из-за вталкивания(выталкивания) в(из) стек(а) (ср. раздел 4). Он считает, что в данном случае выгоднее использовать динамическую память (ср. [8, 9]). Подход Аппеля заключается в многопоточной реализации с «копирующим сборщиком мусора» [6], который действует только тогда, когда было совершено изменение в данных, а зарезервированный объём превышает в определенное число раз объём свободных heap. Несмотря на возраст работы [10], разделение памяти на быстрый кэш и медленную большую память остаётся актуальным. Ларсон рассматривает «уплотняющий сборщик мусора», работа которого зависит от объёма освобождаемой области (R), объёма активных данных (A) и объёма быстрой памяти (H). Ларсон постулирует две оптимальные стратегии (для быстродействия) для выделения/очистки динамической памяти:

— максимизация R, если A $\ll$ H не соблюдается;
— приравнивание R:=H, если A $\ll$ H.

Кроме упомянутых в [7] ограничений, существует еще одно, связанное с адресацией. Если речь идёт о «XOR»-связанных структурах данных (описываемых в [11] и [12]), то мусор не может быть локализован классическими методами, описанными в [6]. Адреса подлежащих утилизации объектов (как, например, поля структур или ссылки на связанные элементы в списке) получаются не абсолютными, а относительными.

Если предположить, что A_{XOR} — конечное линейное адресное пространство, то $(A_{XOR}, \oplus)$ определяет группу с известными правилами ассоциативности, нейтральным и обратимым элементом, замкнутую по операции $\oplus$ [13]. Таким образом, можно вычислять заданный адрес динамической памяти в прямом и обратном направлении.

Допустим, даны адреса a и b, тогда $a \oplus (a \oplus b) \equiv b$, а также $(a \oplus (a \oplus b)) \oplus (a \oplus b) \equiv b \oplus (a \oplus b) \equiv a$. Применением XOR-вычисления достигается экономия второго указателя с помощью дополнительной арифметической операции [11, 12], что может быть полезно во встроенных системах с ограниченным объемом памяти. Однако примеры, приведенные в [11], содержат ошибки и являются неполными. Исправленные варианты можно найти в [13].

Мейер [9] предлагает осуществлять сборку мусора непрерывно. На однопоточных системах с этим нельзя согласиться, так как это означает сильную деградацию скорости всей системы. Вместо сборки мусора лучше обнаружить причины возникновения мусора и переписать алгоритм так, чтобы мусор не появлялся. В таком случае затраты по утилизации приблизятся к нулю. Для многопоточных систем [6] наглядно показывает, что сборщик мусора в параллельной нити может эффективно работать, если его периодически включать и если имеется реальная эвристика по оценке объёма накопившегося

мусора. Иначе сбор просто не оправдан активным ожиданием поступления мусора или лишних загрузок очисток. Даже во встроенных системах объём памяти растёт, и сборщик мусора всё больше теряет практическое значение. Следовательно, только обнаружение мусора в «горячих местах» оправдает затраты, то есть исключительно в циклах, которые часто загружаются.

Работа [14] предлагает сборку мусора по возрасту, то есть она выполняется в зависимости от момента времени и частоты использования объекта. Если объект используется часто, то он, наоборот, перемещается в более быстрый регион памяти. На основе большого количества экспериментов, проведенных авторами, быстродействие эвристически приближено к оптимуму. Сборка мусора используется по умолчанию в таких языках программирования, как java, C#, во многих функциональных языках программирования и выключена в языках программирования ISO C и C++ [15].

2.2. Некорректная доступность/недоступность памяти

Проблема некорректной доступности памяти проявляется в тех случаях, когда программа может манипулировать указателями, ссылающимися на память, которая еще не является доступной. Либо наоборот. Рассмотрим это на следующим примере:

```
// object1 небыл построен
MyClass object2=new MyClass();
object2.ref=object1;
// object1 строится
```

В этом примере имеется ссылка на `object1`, хотя в данный момент экземпляр ещё не построен. Эта ошибка не выявляется при втором присвоении, а возникает лишь в месте первого использования переменной `object2.ref`.

2.3. Неверный доступ к памяти

В данных примерах нас прежде всего интересует неверный доступ к динамической памяти. Допустим, некоторый экземпляр объекта `object1` инициализирован, но поле `ref` во время доступа равно `NULL`.

```
// object1.ref==null
value = (object1.ref).attribute1;
```

Тогда, в лучшем случае, программа выйдет из строя, указав на нарушение неверного доступа к сегменту heap. При верификации входная программа до запуска проверяется на ошибки. Если не может быть выявлено, что данное поле всегда инициализировано, то это является признаком ошибки: возможно, имеются неинициализированные объекты, а этого необходимо избегать.

Также проблема неверного доступа возникает в связи с неверной (например, при произвольной) адресацией динамической памяти. Если доступ осуществляется строго в соответствии с полями имеющегося класса, то необходимо выявить, всегда ли доступ во время запуска определён. Если доступ опирается на арифметику указателей (то есть адрес вычисляется в момент исполнения программы, например по таблице), то задача проверки динамической памяти не определена. Это неудобно с практической точки зрения. Но выразимость сильно не страдает, если заранее использовать только известные

поля классов (ср. [16]). В общем случае для исключения неверного доступа требуется мониторинг всех частей динамической памяти.

Рассмотрим следующий пример.

2.4. Отклонения от структуры данных

В данном примере некоторый объект выводится в виде строки на консоль.

```
object1.next=object1;
...
root=object1;
while(root.next!=null){
  printf(%d, object.data);
  root=root.next;
}
```

Если имеется линейный список с определённым началом и концом, то программа отработает корректно. Однако если структура данных имеет циклическую зависимость, то вычисление может оказаться неверным по отношению к спецификации.

Классический материал по теории объектов при объектно-ориентированном подходе можно найти в работах [17, 18]. Там определяется аксиоматическая база для формальных методов верификации для объектов классов. По Абади и Карделли, объекты являются экземплярами классов, которые имеют различные интерпретации для подтипов одного класса.

В [19] объекты рассматриваются не как абстрактные типы данных, а как обыкновенные записи [20]. Абади не вводит рекурсивно определенные классы и вообще не обращает внимания на указатели и объекты псевдонимов (ссылки). В предлагаемой модели объекты всегда представляются непересекающимися областями динамической памяти. Типы классов (T) определены рекурсивно: они могут быть целым числом, либо составным типом, то есть классом, который содержит множество других типов. Таким образом, объект класса содержит множество отличающихся друг от друга полей и методов. Поля имеют тип T, а методы имеют тип $T_j \to T_{j+1} \to \cdots \to T_k$. Проверка, является ли объект экземпляром определённого класса, и проверка на принадлежность подклассу определяются как покомпонентное сравнение полей и методов.

Состояние «до» и «после» программной команды описывается с помощью одного регистра результата. Рекурсивные предикаты запрещаются [21], так как имеются твёрдые теоретические ограничения данной модели. В работе [22] делается попытка ослабить данное ограничение посредством алгебраической конструкции «кольцо идеала». Однако бывают случаи, когда верификация может оказаться некорректной из-за многозначности при рекурсии, которая реализуется комбинатором для объектов. Это является серьезным ограничением для применения на практике. Более того, модель не разделяет heap и имеет ещё одно фундаментальное ограничение: неполноту (ср. раздел 3). Банерье в [23] представляет язык, который поддерживает чисто стековые объекты одного и того же региона памяти (ср. [8, 24]). Подход в [23] перемещает все локальные переменные в стек (см. раздел 4), но висячие указатели не допускаются по определению языка. Рекурсивные предикаты над объектами запрещены. Особенность глобальных инвариантов заключается в неменяющихся зависимостях между объектами. Банерье предупреждает о сильно возрастающей проблеме абстракции и поддерживает инициативу для пообъектного просмотра индивидуальных предикатов.

В работах [25, 26] делается попытка ввести систематику, основанную на методологии моделирования объектно-ориентированной парадигмы «*Unified Modeling Language* (UML)» [27]. Кроме графического моделирования имеется расширение «*Object Constraint Language* (OCL)» [28], которое представляет собой текстовую запись. Формулы на OCL описывают объекты классов и их взаимосвязи. Выразимость в данном случае эквивалентна типизированному λ-вычислению второго порядка (например в системе верификации «System F»). Однако, «UML» и «OCL», которые являются лишь языками моделирования, не в состоянии выразить псевдонимы и указатели на объекты, которые находятся в динамической памяти, что является большим недостатком.

2.5. Удаленная манипуляция объектами и псевдонимы

Следующий пример демонстрирует два аспекта. Во-первых, `const` не обязательно защитит содержимое от манипуляции извне, даже если запретить присвоения, как это было сделано с указателем pa2 в следующем примере:

```
int *x;
int *good() { int *p = x ; return p; }
int *bad()  { int x = 55; return &x; }

void main(){
  int a=5, b=4;
  int *pa = &a;
  const int *const pa2 = &a;
  a = 77;
  pa2 = pa;
  /***  ' pa2 ' is read - only ***/
  printf( " *pa:%d, *pa2:%d \n" ,*pa, *pa2);
  /***  *pa==77, *pa2==77    ***/
  a = *&b;
  /***  *const of pa2 is still no guarantee for safety   ***/
  printf(" *pa:%d, *pa2:%d\n", *pa, *pa2);
  /***  ! problem: *pa==4, *pa2==4   ***/
  x = (int*)malloc(10);  memset(x,7,sizeof(x));
  printf("0x%x\n", *(good())); // OK
  printf("0x%x\n", *(bad())); // SEGV
}
```

Во-вторых, цикл жизни переменных и содержимого, то есть динамической памяти, кардинально отличается от стековых переменных. Таким образом, правильное присвоение адреса локально внутри процедуры верно. Однако доступ после выхода из процедуры по этому же адресу, скорее всего, приведёт к системной ошибке, что продемонстрировано функциями good и bad.

Определение псевдонимов является на данный момент частично открытой проблемой. Когда два указателя ссылаются на одну и ту же ячейку динамической памяти, то один из указателей является «псевдонимом» другого указателя. В межпроцедурных вызовах имеются «вызываемая» (внутренняя) и «вызывающая» (внешняя) сторона. Алгоритм поиска всех псевдонимов внутри процедуры можно построить относительно просто. Для этого существуют эффективные алгоритмы (например алгоритм Мучника [29]).

Анализ межпроцедурных псевдонимов в самом общем случае может оказаться довольно трудным за счёт необходимости анализа всего вызывающего контекста [30], то есть всех выходящих переменных, которые либо могут (либо не могут) поменяться после вызова рассматриваемой процедуры. Неспособность отличать псевдонимы от «непсевдонимов» является главной причиной неэффективного анализа. На практике, в «GCC» и «LLVM Clang», есть возможность различать псевдонимы с помощью вспомогательных опций, например `-fstrictaliasing`. Также с помощью ключевого слова `__restrict__` поддерживаются переменные-указатели, строго не являющиеся псевдонимами. Данное ключевое слово может быть использовано для исключения псевдонима на уровне объектных методов; достаточно приписать его в качестве атрибута метода (либо при объявлении указателя) [31].

Вайль [3] представляет широкий обзор по тематике псевдонимов, несмотря на возраст статьи (не упоминая работу Мучника; см. далее). По Вайлю анализ псевдонимов является побуждающим процессом приближения указателей, ссылающихся на одну и ту же ячейку памяти, создающую множество псевдонимов с экспоненциальным ростом вариантов и поэтому имеющую возможность быть измененной. Анализ псевдонимов за пределами процедуры гораздо тяжелее, потому что необходимо анализировать все возможные вызовы и продолжения, в которых переменные будут существовать. Естественно, в продолжениях переменные также могут меняться, что и является причиной затрудненного анализа. Чтобы упростить анализ, Вайль вводит ярусы псевдонимов, по которым оценивается сложность каждой программной команды.

Мучник [29] представляет полный обзор всех нынешних и прошлых методов анализа псевдонимов и свои собственные в том числе. Мучник делит методы на зависимые от графа потока управлений и не зависимые от него, на «псевдонимы» и «возможно, псевдонимы», а также на замкнутые подпрограммы и вызовы подпрограмм.

Горвиц [32] расширяет и ускоряет подход Мучника с помощью битовых векторов. Обобщенные подходы анализа следует также искать в [33], где предлагается представление потока управления данными в качестве битовых векторов. Векторы обновляются с помощью обобщённого алгоритма Флойда-Воршалла для достижения всех вершин в направленном графе [34]. Более того, мы считаем, что представленный подход анализа присвоений и использований нужно обобщить в будущем, используя «SSA»-форму [35, 36].

Наим [37] подтверждает наше личное мнение. Он видит потенциал к улучшению доступа, например, с помощью хэш-таблицы. Стоит отметить, что подход Наима содержит предпосылки и намерения переписать проблему анализа псевдонимов как проблему «SSA». Остаётся открытым вопрос, как это сделать? Такой подход позволил бы сильно сократить одну фазу статического анализа, при этом увеличить производительность за счёт полного и эффективного обращения к регистрам, избегая безопасных и медленных шагов копирования целых структур данных.

Павлю [38] делит методологии анализов псевдонимов на две категории: подход с помощью «унификации» [39], который находит больше случаев «псевдоним», и на подход с помощью «плавления», который слабее из-за скорости анализа. Подход с помощью «плавления» всё равно является компромиссным. Вопрос, можно ли бескомпромиссно и эффективно исключать псевдонимы явным образом и переписывать последовательности инструкций, требующих удаленной манипуляции с памятью, остается открытым. Другими словами, интересно рассмотреть сценарии, когда псевдонимы не исключаются полностью, но возможно изменить структуру программы так, чтобы модифицируемая

часть памяти была заключена локально внутри процедуры. Нам кажется, что этот вопрос во всей имеющейся на данный момент литературе освещён недостаточно.

3. ИСЧИСЛЕНИЯ ХОАРА

Первой статьей посвященной спецификации и верификации является работа Хоара 1969 года [2]. Обратим внимание, что, несмотря на большой возраст, упомянутые в этом разделе статьи практически без изменения сохраняют актуальность из-за фундаментального характера теоретических проблем. В статье Хоар предложил формализм для описания семантики языков программирования на основе логических правил и аксиом. К тому времени уже были предложены денотационная и операционная семантики. Аксиоматическая семантика по Хоару исходит из спецификации, которая описывает, какое состояние вычисления ожидается в определенный момент исполнения программы. Оно сравнивается с актуальным состоянием вычисления. Если состояния совпадают, то программа выполняет данную спецификацию, иначе программа не удовлетворяет требованиям корректности. Хоар предлагает в качестве формализма алгебраические и логические утверждения. В честь него была названа «Тройка Хоара» $P\{C\}Q$, которая сегодня чаще всего записывается как $\{P\}C\{Q\}$, где P предусловие, C программный оператор, а Q является постусловием.

Обозначим Γ — набор логических правил формы $\frac{A}{B}$, где A антецедент, а B консеквент. Аксиомой является правило, чей антецедент — тавтология, согласно рассматриваемому дискурсу, и записывается как $\frac{}{B}$, либо как $\texttt{true} \to B$. Если данная тройка Хоара синтаксически выводима из Γ, то мы записываем это как $\Gamma \vdash \{P\}C\{Q\}$ или как $\vdash_\Gamma \{P\}C\{Q\}$. Если из контекста ясно, что используется только Γ, то $\vdash_\Gamma$ упускается ради простоты. Тройка Хоара интерпретируется как предикат, который определён лишь в том случае, когда C терминирует (является терминальным). Интерпретация: если предположить, что предусловие P верное, выполнен оператор C и постусловие Q также соблюдается, тогда тройка верна. В противном случае, тройка неверна. В случае нетерминации C тройка неопределена. Неверность тройки может происходить из-за несоблюдения постусловия. Постусловие устанавливается разработчиком или тестировшиком. Если Q — неверно согласно данной программе, то и результат вывода вызывает сомнения. P и Q задаются формулами, которые описывают состояние вычисления, то есть должна иметься связь между переменными среды, переменными и символами в C. Хотя Хоар не утверждал этого, но подразумевается использование формальной логики утверждений для описания языков программирования.

Уже в 1969 году Хоар выявил, что некоторые C достаточно просты для описания (и, в итоге, для выполнения верификации), а некоторые существенно более сложны. Ко второй категории Хоар относит метки, безусловные переходы, произвольные параметры. Такое состояние дел имеет место и сегодня. Метки часто исключаются из современной верификации полностью из-за модульного принципа построения программного обеспечения, в котором использование безусловных меток не приветствуется, однако это касается только прикладного программного обеспечения, написанного на языках высокого уровня. Что касается произвольных параметров, то тут Апт [40] и Кларк [41] провели анализ, итогом которого можно считать решение этой проблемы с практической точки зрения для Алгол-подобных императивных языков программирования.

Доказательство проводится с помощью применения данных правил до тех пор, пока не будет установлено противоречие (то есть имеется доказательство, что данная про-

грамма не корректна), либо выведены все аксиомы (то есть доказательство успешно). Структура доказательства всегда является деревом. Если имеется цикл, то доказательство не осуществимо.

Доказательство начинается с консеквентом и ищет имеющиеся условия в антецеденте для каждого из настоящих утверждений до тех пор, пока не будет сведено к аксиомам. При этом C делится до тех пор, пока изначальная программа не будет состоять только из неделимых программных операторов. Необходимо отметить, что данный язык программирования не был определён Хоаром, однако можно видеть, что он имеет императивный характер. Здесь выявление состояния вычисления (выполнения программы) может быть выполнено последовательно, а значит, может быть предложен дескриптивный язык программирования. Кроме специфических функций каждого языка программирования, имеются и общие сложности в определении верности и применимости правил верификации. Одним из таких правил является проблема предсказания инварианта циклов. Рассмотрим правило цикла:

$$\frac{\{p \wedge e\}\ S\ \{p\}}{\{p\}\ \texttt{while}\ \ e\ \ \texttt{do}\ \ S\ \ \texttt{od}\ \{p \wedge \neg e\}},$$

где e представляет условие, S — блок программных операторов, p — предусловие. В консеквенте очевидно, что при выходе из цикла начальное условие e должно быть ложным, однако все утверждения, объединившись в p до цикла, должны быть верными и после запуска цикла. Чтобы доказать верность данного цикла, необходимо лишь доказать верность тела цикла с p и начальным условием e, так как тело цикла в принципе доступно только до тех пор, пока e верно. Однако после исполнения S e может быть верным или неверным.

Для полного определения цикла необходимо включать все переменные, которые используются в цикле, и определять необходимые равенства и неравенства с помощью формул. При исполнении цикла состояние вычисления меняется, и необходимо задуматься о параметризации в зависимости от меняющихся параметров. Формулы, которые зависят от этих параметров, как раз и являются «инвариантами». Очевидно, что определение инвариантов не всегда может быть выполнено полностью автоматически для любой входной программы из-за теоретических ограничений, например для многопоточных приложений, где имеет место приостановка выполнения и асинхронная смена состояния.

Что касается минимальной входной программы, то во многих подходах наблюдается рационализация к следующему минимуму:

$x := t$	.. присвоение
$S_1; S_2$	.. оператор последовательности
`if` e `then` S_1 `fi`	.. условный переход
`while` e `do` S`od`	.. цикл

Строго говоря, условный переход также может быть сопоставлен циклу. С теоретической стороны программа минимальна, однако, в связи с выразимостью и провождением верификации, разные модусы переменных уже составляют фундаментальную проблему для представления в качестве троек Хоара [41].

Апт [40] спустя десятилетия после Хоара провёл полный анализ подходов исчисления Хоара и подтвердил тенденцию в сторону минимальных программ. Апт в [40, 42] показывает, почему возникают некоторые «неприятные» свойства выразимости и полноты,

которые практически без изменения остаются в силе до сегодняшнего дня. Это происходит, например, из-за произвольной рекурсии, которая в состоянии менять полностью состояние вычисления, из-за рекурсивно-определённых структур данных, которые вычисляются частично только при доступе к их полям, а также из-за процедур в качестве параметров.

Кук [43] предлагает ограниченное подмножество Хоара и Апта [40] ради исключения произвольной рекурсии и не разрешает сопроцедуры и такие процедуры, как параметры и иные неалголовские конструкции (например ленивые структуры данных). С помощью абстрактной машины, соответствующей операционной семантике, он доказал, что следующий набор правил верификации, отличающийся минимально от Хоара, является полным (по Куку) и корректным:

$$\text{VARDECL}\ \frac{P[y/x]\{\texttt{begin}\ D^*;\ A^*\ \texttt{end}\}\ Q[y/x]}{P\{\texttt{begin}\ \ \texttt{new}\ \ x; D^*; A^*\texttt{end}\}Q} \qquad \text{PROCDECL}\ \frac{D,\ P\{\texttt{begin}\ D^*;\ A^*\ \texttt{end}\}Q}{P\{\texttt{begin}\ D; D^*; A^*\ \texttt{end}\}Q}$$

$$\text{COMP1}\ \frac{P\{A\}Q,\ Q\{\texttt{begin}\ A^*\ \texttt{end}\}R}{P\{\texttt{begin}\ A; A^*\ \texttt{end}\}R} \qquad \text{COMP2}\ \frac{}{P\{begin\ end\}P}$$

$$\text{ASN}\ \frac{}{P[e/x]\{x := e\}\ P} \qquad \text{COND}\ \frac{P \wedge R\{A_1\}Q,\ P \wedge \neg R\{A_2\}Q}{P\{if\ R\ then\ A_1\ else\ A_2\}\ Q}$$

$$\text{WHILE}\ \frac{P\ \wedge\ Q\ \{A\}\ P}{P\{while\ Q\ do\ A\}\ P\ \wedge\ \neg Q} \qquad \text{CALL}\ \frac{p(x:v)\ proc\ K, P\{K\}Q}{P\{call\ p(x:v)\}Q}$$

$$\text{VSUB}\ \frac{P\{call\ p(u:e)\}Q \qquad \sigma = z'/z}{P\sigma\ \{call\ p(u:e)\}\ Q\sigma} \qquad \text{CONSEQ}\ \frac{P > R,\ R\{A\}S,\ S > Q}{P\{A\}Q}$$

$$\text{PSUB}\ \frac{P\{call\ p(x:v')\}Q}{P\ u,e/x',v'\ \{call\ p(u:e)\}\ Q\ u,e/x',v'}$$

Кук замечает, что наиболее практическими преградами являются: (1) определение нетерминации не может быть вынесено с помощью или внутри исчисления Хоара в общем случае, (2) выразимость языка утверждений (то есть спецификации и верификации) — одна из самых главных проблем, которые нужно преодолеть. Можно также вывести, что доказательства необходимо упрощать, потому что «простое» доказательство является хорошим, и нужны хорошие доказательства, когда речь идёт о проверке свойств программ. Представим себе, что данная программа не совпадает с данной спецификацией. В таком случае необходимо без больших затрат определить, в чём заключается нестыковка и какой имеется противоречивый случай. Конечно, замена параметров имеет недостатки и может привести при неудачном выборе к циклу, но это не решающий вопрос в предложенном наборе правил. Однако вопрос о сходимости (с англ. «*proof confluency*») является важным, всё ещё не решённым вопросом. Под сходимостью доказательства подразумевается применение логических правил по порядку так, чтобы всегда был выводим один и тот же результат.

Кларк [41] называет самыми важными проблемами исчисления Хоара те же, что Кук и Апт. Он обозначает проблемы, не решённые и в принципе не решаемые в исчислении Хоара. Такими являются продолжения, модусы переменных (статические, автоматические, динамические и глобальные). Проблемы можно считать старыми, глядя на год публикации, однако, изучая появившиеся позже подходы, можно констатировать, что ни-

чего фундаментально не поменялось. Например, статически выделенные переменные трудно вписываются в исчисление Хоара из-за необходимости избегания процедуры выталкивания элементов из стека. Определение полноты по Кларку не отличается от определения по Куку. Под полнотой оба автора подразумевают то, что каждая верная формула доказуема. Кларк выявляет в дополнение к Апту [40] «плохие» свойства, ломающие полноту. Самоприменение в рекурсиях в связи с неопределенностью Кларк рассматривает как возможную уязвимость корректности и полноты. Вспомним, что любая формальная система по Гёделю неполна, так как всегда можно сформулировать высказывания, не доказуемые средствами правил той же самой формальной системы. Это безусловно касается и исчисления Хоара, в частности, доказательства динамической памяти. Крайние случаи важны с теоретической точки зрения, хотя с практической точки зрения они крайне незначимы, так как в большинстве случаев верификация может проводиться. Это же касается принципиальной решимости термов при верификации. Они ограничиваются лишь сложением / вычитанием (так называемая арифметика Пресбургера [44]), но исключают умножение и другие операции. Однако теоретическая разрешимость всё равно на практике не достаточна из-за экспоненциальной сложности в худшем случае. Здесь практическая реализация и теоретические границы сильно расходятся. Теоретические границы для практической реализации малозначимы с точки зрения ежедневной применимости.

Кларк исследовал комбинацию признаков программ, которые допускают и исключают полноту. Например, если имеется императивный язык программирования с:

(i) процедурами в качестве параметров,
(ii) произвольными (то есть (μ-рекурсивные) процедурами,
(iii) статическими переменными,
(iv) глобальными переменными,
(v) иерархически содержимыми процедурами,

то полнота в общности может нарушаться из-за iii. При исключении проблемных свойств доказательства будут полными, если придерживаться i-ii и iv-v. Доказательство этого приводится Кларком с помощью реляций соответствующей операционной семантики и, по сути, близко к соотношению эквивалентности по Куку. Наблюдения о процедурах далее могут быть полезными в связи с логическими предикатами в подходах верификации динамической памяти и абстрактными предикатами. Кларк показывает, что исключение самоприменимых параметров процедуры ограничивает выразимость, но, с другой стороны, приводит к вычислению без неожиданных некорректных переходов. Сопроцедуры в комбинации с произвольной рекурсией могут привести к неполноте и некорректности.

В [45] Мейер отмечает, что переменные, которые выделяются в произвольном порядке, резко усложняют спецификацию и верификацию куч. Но их устранение нельзя рассматривать как решение проблемы, так как их ограничение эквивалентно существенному снижению выразимости. Далее Мейер утверждает, что, на первый взгляд, присваивание $a := b$ корректно, согласно его правилу Хоара для присваивания. Однако опущен один важный сценарий: когда производится вызов процедуры в b, то, потенциально, всё состояние вычисления без ограничения общности может поменяться и стать некорректным. Согласно продемонстрированным подходам Апта, Кларка и Кука, нельзя рассматривать корректность по отношению только к одному правилу, необходимо всегда включать целое множество правил [41]. Множественное объединение правил $\bigcup$ в общности не замкнуто касательно корректности.

По мнению всех ранее перечисленных авторов, разрушающим критерием являются переменные с динамической памятью. Получается, что не только Миллер [5] с практической стороны, но также Кларк и другие авторы, независимо друг от друга, подтверждают, что работа с динамической памятью является крайне тяжелой. Кларк и все другие, ранее упомянутые авторы, предлагают дальнейшее исследование динамических переменных, но утверждают, что это является трудной проблемой в исчислении Хоара.

4. ПРЕОБРАЗОВАНИЕ КУЧИ В СТЕК

Подход «преобразовать в стек» часто можно встретить в литературе [6, 7, 9, 45, 46]. Мотивируется такой подход тем, что указатели тяжело анализировать. Эта причина только упоминается, хотя является важной причиной (см. раздел 2), и сбор мусора требует дополнительных затрат. Однако Аппель наглядно проиллюстрировал, что это далеко не всегда является проблемой [7]. Точнее, бывают ситуации, когда сбор мусора быстрее работы стека. Безусловно, работа с указателями может быть непростой. Однако алгоритмы могут быть очень простыми и очень быстрыми по сравнению со стековой реализацией. Почему быстродействие также имеет немалое значение? В случае не разрушаемого копирования данные дублируются, и любой алгоритм тратит основное время на чтение и запись памяти из-за архитектуры фон-Неймана. Даже при разрушаемом копировании, что часто встречается на практике (хотя анализы самых простейших примеров на основе GCC 6.3.0 [31] заставляют ожидать худшего), структуры из-за конвенций «Application Binary Interface» всё равно могут запретить более эффективное использование, особенно при передаче параметров в процедурах. Таким образом, эффективное использование просто блокируется изначально. GCC разбивает объекты и структуры на процессорные слова. Это происходит в основном без соблюдения принадлежности классового экземпляра, что на самом деле не является проблемой на этапе генерации кода. Однако до сегодняшнего дня, если экземпляр объекта должен был быть удалён, то, к большому сожалению, этого часто просто не происходило, вследствие чего программный код «раздувался». В частности, связанные между собой объекты далее существенно уменьшали шансы на минимальный код. Если все экземпляры вталкивать / выталкивать в / из стек(а), то это может быть решено одновременно, однако это происходит за счёт полного копирования объектов. Если имеются обновления в иерархии вызовов методов, то объект не должен уничтожаться, а должен передаваться согласно уровню вызова.

Преобразование кучи означает, что структура данных из динамической памяти должна помещаться полностью в стековое окно. Зависимости должны кодироваться неявным образом. Например, если имеется линейный простой односвязный список, то все последующие элементы можно поместить по возрастающему индексу. То же самое можно произвести с двусвязным списком. Если элемент вставляется или удаляется из указанного места списка, то стековая структура данных может сильно поменяться, так как требуется учесть перемещение для соблюдения правильного порядка. То же самое необходимо соблюдать при «стекизации» деревьев и графов.

Мейер предлагает хранить все объекты в стеке. Аппель (и не только он) показал, что это, с точки зрения быстродействия, может быть избыточно. Замена алгоритма на такой, который работает только со стеком, показывает не очень хорошие характеристики для стандартных структур данных. Например, выделение стека оператором `calloc` вместо `malloc` может занять несколько килобайтов на стеке. Нельзя не учесть, что из-за этого стековое окно может сильно разрастаться. Как следствие, страницы операционной си-

стемы должны будут чаще распределяться, что отрицательно скажется на общих характеристиках скорости исполнения программы (см. «variadic functions» [15]).

Успешность стратегии помещения всех объектов в стек не ясна априори, наоборот, имеется большое подозрение в ее неэффективности. Динамические структуры данных используются в основном тогда, когда изначально не известен их размер. Часто возможность манипуляции с динамически выделенными данными описывается в стеке с большими ограничениями или неэффективной реализацией (например, медленность или слишком большие объемы запасной памяти, негибкость реализации). Поэтому с перемещением в стек так, как это предлагает Мейер, согласиться сложно. Справедливости ради отметим, что программы, использующие динамическую память, тоже можно написать как простым, так и сложным образом, аналогично алгоритмам со стеком. По Мейеру, анализ указателей и псевдонимов усложняется ещё потому, что выбранная им логика не делает никаких различий между структурами динамической памяти и косвенными неструктурными, а именно логическими высказываниями. Это означает, что существенные усилия должны быть потрачены на определение области дискурса.

Частным случаем преобразования куч в стек является так называемое «вычисление регионов» [8]. Главная идея заключается в том, что объектные экземпляры динамической памяти присваиваются как локальные переменные, то есть как объекты, размещённые на стеке, и выделяются / уничтожаются при входе в блок и выходе из него. Размер экземпляра известен. Приближение подклассов может быть совершено, если присвоить каждому подтипу максимальный возможный размер. Такой подход представления более распространён в функциональных языках программирования, в частности, в «ML». В связи с функциональным подходом имеются ограничения: нельзя возвращать списки, диапазон видимости необходимо отдельно моделировать, так как диапазоны символов в функциональных языках не ограничиваются блоками. Еще одна слабость заключается в том, что регионы, подлежащие утилизации и долго находившиеся в стеке, с какими бы то ни было более актуальными объектами практически никогда не связаны (очень редко). При этом анализ именно для таких категорий значительно усложняет и тормозит весь процесс проверки.

5. АНАЛИЗ ОБРАЗОВ

Главной целью анализа образов [30, 38, 47, 48] является выявление инвариантов в кучах памяти, например для обнаружения псевдонимов. Граф зависимостей между образцами описывается всегда полностью с помощью трансферных функций, таких как образ «пуст», присвоение значения полю или указателю и выделение новой динамической памяти. Сагив [47] предлагает категоризацию соотношений между двумя указателями: «псевдонимы», «не псевдонимы» и «возможно, псевдонимы». В сжатых подграфах зависимостей наименование вершин остаётся общей проблемой в [47] и [48]. Эти подходы имеют целый ряд ограничений, как, например: доступ к указателям с индексом разрешает лишь не переменные выражения, исключаются объектные указатели, запрещаются массивы с гибкой шириной, исключаются пересекаемые в памяти структуры данных (как, например, «union»-структуры на языке Си).

В [38] Павлю замечает, что [47] и [48] могут привести, в зависимости от данной программы, к неточному и, следовательно, к неправильному выводу. Если для «если-тогда»-команды в одном случае вычисляется «возможно, псевдонимы», а в альтернативном случае «псевдонимы», то результатом вычисления послужат «псевдонимы», хотя правиль-

ный ответ «возможно, псевдонимы». Более того, [38] содержит подробное сравнение подходов [47] и [48]. Павлю [38] оценивает [48] как более точный метод и предлагает оптимизацию путей по графам образов с одинаковыми началами и псевдонимами. Предложенная оптимизация имеет квадратный верхний порог сложности и сокращает вычисление в среднем случае примерно на 90%. Павлю предлагает симулировать более сложный анализ псевдонимов за пределами процедур, преобразуя, насколько это возможно, вызовы процедур и глобальные переменные в локально интрапроцедуральные элементы через переименование. Подход, используемый им, был на самом деле уже ранее успешно применён в системе «GCC» для решения комплекса иных проблем. Павлю, сравнив полученные характеристики, так же, как и мы, считает, что контекст-независимые подходы в анализе псевдонимов малоперспективны (ср. [49]). Далее он предлагает оптимизацию отделения объединяющих вершин, которые образуют подграфы, от вершин, которые точно не содержат псевдонимов. Пардун [50] предлагает среду для визуализации образов куч для быстрого анализа и обнаружения инвариантов, редко используемых связей между образами. Для навигации по графу используются операции «абстракция графа» и «конкретизация графа», с помощью чего удаётся подграфы свёртывать и развёртывать. Визуализация трансферных функций, которая приводит к развёртыванию / свёртыванию более одного подграфа, оставляет желать лучшего, однако принципиальных улучшений не ожидается.

Калканьё [51] даёт собой подход, основанный на распределении куч по образцам (см. раздел 7). Этот подход сравнивает предположительно подходящие начала правил по длине и выбирает наиболее длинное правило первым. Особенность подхода заключается в аппроксимации обеих сторон рассматриваемых правил для выбора принимаемых правил с помощью ограниченной абдукции.

6. РОТАЦИЯ УКАЗАТЕЛЕЙ

Старая, но не устаревшая теория «ротация указателей» [52] предлагает операции ротации и трансляции над указателями. Например, если предположить, что имеется дизъюнкция указателей, то одна допустимая бочечная ротация может быть определена как пермутация (где, например, x_1 указывает на x_2):

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots x_{n-1} & x_n \\ x_2 & x_3 & x_4 & \cdots x_n & x_1 \end{pmatrix}$$

Трансляция влево `slide` будет конгруэнтна левой бочечной ротации. Далее `x:=y` эквивалентна $slide(x, y)$. Цель ротаций и трансляций заменить малоизвестный потенциально опасный код на хорошо известный правильный код (в этом случае используемые надёжные функции над указателями выступают в качестве спецификации). Например:

```
y=NULL;
while (x!=NULL) {
  temp = x.next; x.next = y; y = x; x = temp;
}
```

будет эквивалентно следующей записи:

```
y = NULL;
while (x!=NULL) rotL(x,x.next,y);
```

Не сложно из данного примера не из статьи выявить, что `rot(x,x,y)` тождественное отображение id. Аналогично предыдущей правой ротации можно определить левую ротацию $rotate()$:

```
y = NULL;
while (x!=NULL){
  temp = x;
  x = y;
  y = x.next;
  x.next = temp;
}
```

Для этой ротации можно теперь установить равенство:

$$rot_L^{-1}(x, x.next, y) \equiv rot_L(x.next, x, y).$$

Таким образом можно определить и доказать алгебраические свойства операции в зависимости от задаваемых трансляций и ротаций. Одна и та же ротация имеет, в зависимости от входных данных, разные стереотипы. Например: $rotate(x.next, x.next.next.y)$. Здесь y является целевым списком, $x.next$ является первым аргументом, который будет сдвинут, а $x.next.next$ вторым аргументом, оставшимся указателем, который не подлежит исправлению.

Этот подход определённо интересен с теоретической и практической сторон. Однако само вычисление с помощью ротаций мало пригодно для исключения сложных алгоритмов над деревьями, тут требуется исследование фактора пригодности.

Ротация указателей (включая трансляции) считается безопасной по определению. Это далеко не так. Например, $rotate(y, y.next)$ удаляет первый элемент в некоторых конфигурациях, а $rotate(x, y, x.next)$): если (1), — содержание куч не меняется, (2) — все элементы ротации годны до и после ротации, (3) — количество переменных не меняется. Преимуществом безопасной ротации можно считать не только отсутствие нужды сбора мусора, который просто определяется, но также эффективные и корректные операции над списками, как, например, копирование. Хотя ротация указателей не нуждается в явной спецификации, всё-таки минимальное изменение параметров может привести к трудно прогнозируемому поведению программы (особенно часто, когда указатели неожиданно оказываются псевдонимами). Сузуки [52] предлагает «базисные» ротации (определение термина «базисная» при этом умалчивается). Однако на практике этого далеко не достаточно, поэтому также необходимо комбинировать индивидуальные ротации из базисных.

7. ЛОГИКА РАСПРЕДЕЛЕННОЙ ПАМЯТИ

Логика распределённой памяти (ЛРП) использует предикаты для описания динамической памяти [53–56]. Для этого динамическая память описывается пространственными операциями (оператор «$\star$»), а логические — логическими конъюнкциями (операторы «$\wedge, \vee, \neg$»).

$$\begin{aligned} \Phi \quad ::= \quad & \underline{true} \mid \overline{false} \mid x \mid REL(f_j(\vec{x})) \mid Pred(f_j(\vec{x})) \\ & \mid \neg\Phi \mid \Phi \star \Phi \mid \forall x.\Phi[x] \mid \exists x.\Phi[x] \end{aligned}$$

В частности, $REL(f_j(\vec{x}))$ включает в себя базисную кучу $a \mapsto b$, где a локация, а b допустимое значение (возможно, объектного класса). Локация означает либо идентификатор, либо выражение доступа по полям согласно определению полей объектного экземпляра. Однако в начальном определении логики распределённой памяти нет возможности описывать объекты; она вводится позже Паркинсоном [57]. $Pred()$ и обозначает вызов заранее определённого предиката, который может иметь параметры.

Пространственный оператор по Бёрдайну и Рейнольдсу подразумевает делимость между частями куч. Строго говоря, оператор «$\star$» не только может делить (как было предусмотрено авторами), но также может быть использован в качестве объединения. Объединение существенно усложняет анализ куч и всех подграфов.

Свойства пространственного оператора по Рейнольдсу [53] и по Бёрдайну [54]:

(1) несжимаемость:
$p \not\Rightarrow p \star p$, $p \star q \not\Rightarrow p$, если $\exists q, q \not\equiv \mathsf{emp}$

(2) коммутативность:
$p_1 \star p_2 \Leftrightarrow p_2 \star p_1$

(3) ассоциативность:
$(p_1 \star p_2) \star p_3 \Leftrightarrow p_1 \star (p_2 \star p_3)$

(4) нейтральный элемент:
$p \star \mathsf{emp} \Leftrightarrow \mathsf{emp} \star p \Leftrightarrow p$

(5) дистрибутивность:
$(p_1 \vee p_2) \star q \Leftrightarrow (p_1 \star q) \vee (p_2 \star q)$
$(p_1 \wedge p_2) \star q \Leftrightarrow (p_1 \star q) \wedge (p_2 \star q)$

(6) квантификация:
$(\exists x.p) \star q \Leftrightarrow \exists x.(p \star q)$, если $x \notin FV(q)$
$(\forall x.p) \star q \Leftrightarrow \forall x.(p \star q)$, если $x \notin FV(q)$

Предикат emp верен, когда передаваемая куча пуста. Множество FV определяет все свободные переменные для данного утверждения. В качестве параметризованного предиката рассмотрим двойное дерево, которое можно индуктивно определить следующим образом:

$$btree(l) ::= \mathtt{nil} \mid \exists x.\exists y:\ l \mapsto x, y \star btree(x) \star btree(y)$$

Двойное дерево, которое передаётся предикату $btree$ в качестве нетипизированного символа l, либо пусто (то есть если данное дерево пусто и соблюдает предикат $btree$, то результат предиката «верно»), либо l ссылается на линейный список, содержимое которого x, а затем y, при этом оба содержимых не пересекаются и имеют различные указатели.

ЛРП основана на исчислении Хоара (см. раздел 3) и является подструктурной логикой [58], которая избавляется от констант, например от булевских значений. Обратим внимание, что верхние константы, например $\underline{true}$, являются частичными предикатами, которые принимают кучу и возвращают булевское значение в зависимости от переданной кучи. ЛРП также использует символы для структурных замещений в качестве констант. В ЛРП структурными правилами служат (согласно правилу повтора (THIN) [58]) сужения (CONTR) и сопоставления, а «константами» служат ячейки динамической памяти. В указанных правилах «,» заменяется на «$\star$», которая разделяет две не пересекаемые и отличные друг от друга кучи, кроме того случая, когда оговаривается что-нибудь другое. Кучи в динамической памяти определяются индуктивно. Правила

(PERMUTE) и (CUT) предусмотрены для вычитания. В ЛРП правило сужения не соблюдается, следовательно, кучи не могут повторяться. Это свойство очень полезное, когда часть динамической памяти может и должна указываться максимально один раз.

$$\text{THIN}\frac{X, Y \vdash Z}{X, A, Y \vdash Z} \qquad \text{CONTR}\frac{X, A, A, Y \vdash Z}{X, A, Y \vdash Z} \qquad \text{PERMUTE}\frac{X, A, B, Y \vdash Z}{X, B, A, Y \vdash Z}$$

$$\text{CUT}\frac{X \vdash A \quad U, A, Y \vdash Z}{U, X, Y \vdash Z}$$

Правило Фреймов:

$$\frac{\{P\}C\{Q\}}{\{P \star F\}C\{Q \star F\}}$$

означает, что если вызов подпроцедуры C не меняет части куч, а именно раму обозначенной F, то в антецеденте достаточно доказать, что тройка Хоара без F верна.

Бёрдайн [54] водит вычисление на основе ЛРП для неограниченной арифметики над смещениями указателей с возрастающими массивами и рекурсивными процедурами. Оно представляет собой попытку определить области динамической памяти рекурсивно с помощью замкнутого объема встроенных правил. Бёрдайн и соавторы задаются также вопросом, не является ли типизация верификацией. Из статьи следует, например, что нерешаемость безграничного использования указателей приводит к неточному определению момента сбора мусора даже для самых безобидных выражений в качестве офсетов памяти и к неточному выбору правил для верификации, которые управляются жадной эвристикой.

Борна [16] предлагает модель, по сути, очень похожую на ЛРП, которую он называет «дальнее разделение» и которая преобразует объекты в массив (ср. с разделом 4). Таким образом, любое поле объекта становится индивидуальным указателем с конвенцией для наименования и идентичности. Для общего определения куч он постулирует, что необходимо использовать предикаты первого порядка. Главной заслугой Хурлина [59] является нововведенный паттерн для верификации доступа к кучам с многими потоками. Если куча «простая», то есть является неделимой, то имеется тривиальный и верный случай. Если куча недоказуема — в этом случае принудительные упрощения не применяются.

Паркинсон [60] представляет объектно-ориентированное расширение подхода ЛРП [53], используя Java в качестве входного языка. Модульность и наследие моделируются с помощью «передачи собственности над вызовами» и «Семейством Абстрактных Предикатов». Модель доступа Борна [16] применяется, так как свойство непрерывности наблюдается у правила фреймов для объектов. Борна замечает, что зависимость между предикатами создаёт порядок вызовов предикатов. Из статьи вытекает, что его предикаты разрешают определить всю динамическую память и стек, но они не разрешают, например, определить любые предикаты первого порядка. Предикаты утверждения, которые сильно отличаются синтаксисом и семантикой от входного языка, не ограничены в типах, однако использование символов имеет целый ряд ограничений, которые связаны с несимвольной реализацией сопоставления переменных символов. Фактически предикаты используются как локальные переменные в императивных языках программирования. Для дальнейшего исследования Паркинсон предлагает рассмотреть вызовы методов из родительских классов, статические поля, интроспекцию объектов, внутренние классы и кванторы над предикатами.

8. ПЕРСПЕКТИВЫ РАЗВИТИЯ ВЕРИФИКАТОРОВ ДИНАМИЧЕСКОЙ ПАМЯТИ

Кроме уже упомянутых тенденций, пробелов в исследованиях и актуальных дебатов, хотелось бы отдельно выделить следующие направления для дальнейших исследований:

1. Обе модели, по Рейнольдсу (обсуждавшиеся, в основном, в этом обзоре) и по Бурстоллу не распознают на уровне записи различные экземпляры объектов. То есть если их содержимое одинаково, то они, по определению, в обеих моделях должны быть идентичными. Однако на практике это не обязательно, поскольку изменения одного объекта могут не затрагивать другой.
2. Ради исключения предикаты заранее запрограммированных тактик, в основном, вручную свертываются и развертываются (заменив нужные переменные). Хаттон предлагает генеративный метод свёртки и развёртывания в области функционалов в функциональных языках программирования. Здесь стоило бы дальше вести исследования с целью автоматизации доказательства, потому что, в связи с использованием генерации, имеются проблемы. Эти проблемы приводят к тому, что требуется ручное вмешательство из-за невозможности правильного использования предиката.
3. Использование «горячего» кода, то есть программного кода, который сам себя обновляет при запуске программы, вообще не рассматривалось и не рассматривается. Некоторые упомянутые авторы считают такое «расширение выразимости» уникальным. Однако, при всем оптимизме, необходимо задуматься, что возможности сильно ограничены. С одной стороны, включается код, чья спецификация, в лучшем случае, лишь известна, с другой стороны, это создаёт дополнительный фактор риска в связи с безопасностью. Выражение на самом деле не увеличивается, когда речь идёт о жестких спецификациях куч, и представленные в обзоре проблемы необходимо исключить с помощью спецификации куч. Медленность из-за необходимости интерпретации кода вообще не рассматривалась в этом вопросе.
4. Важным критерием адаптации верификатора является уровень интуитивности объяснений доказательств. Это число автоматически играет большую роль, как и обобщенная генерация контрпримеров, которая на данный момент, можно считать, полностью отсутствует.
5. В связи со всё новыми моделями и подходами растёт необходимость интеграции. Требуются большие усилия, для того чтобы заставить совместно работать инструменты, выполняющие различные этапы верификации. Основная причина — неадаптируемое и нерасширяемое промежуточное представление.
6. Для сравнения верификаторов куч часто используются только специфические сценарии. Ведь для этого они все, в основном, и были разработаны. Однако для лучшей интеграции и применимости на определённой стадии необходимо провести ревизию, какой верификатор сколько и какие ошибки распознал. Для этого требуется иметь возможность работать с различными верификаторами по единой, но достаточно широкой, кодовой базе.
7. В связи с интеграцией ожидается больше попыток проводить унификацию подходов и методов для соединения преимуществ сильно различающихся подходов.
8. В параллелизации верификации в исчислении Хоара не ожидается прогресса, что не касается многопоточных верификаций различных программ. Также параллелизация ожидается в соседней области исследования при проверках моделей, так как там необходимо дальше ускорять сужение поискового пространства.

9. Проблемы выразимости и полноты остаются открытыми. Модусы переменных остаются актуальным полем исследований как минимум следующие 12 лет (статические, глобальные, динамические).
10. Доддс [61] предлагает дескриптивный язык трансформации в качестве описания динамической памяти. Подход сильно отличается и, на первый взгляд, не применим к императивным языкам программирования. Здесь требуется более широкое исследование в связи с применимостью для диалектов Си.
11. Предполагается, что функциональная тенденция 2010-х годов в области верификации памяти будет наверно переходить на логические парадигмы. Всё больше и больше наблюдается использование и преимущества в использовании соотношений и логических символов вместо функционалов.
12. Предполагается, что, в связи с возрастающей сложностью систем верификации и языков программирования, будет наблюдаться более сильная алгебраизация логических правил. Удобство вычисления станет более важным критерием, чем отдельные функции языков программирования. В частности, прогнозируется, что линеаризация логических выражений разрастётся за пределами динамической памяти и будет охватывать память целиком.
13. Ожидается, что функции высшего порядка в исчислениях Хоара будут играть всё меньшую роль.
14. Частичная спецификация была выявлена как прагматизм в плане сокращения необходимого доказательства объёма программы.

Список литературы

1. *Love R.* Linux Kernel Development. Addison-Wesley Professional, 3rd edition, 2010.
2. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM. 1969. № 12(10). С. 576–580.
3. *Weihl W. E.* Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors // Proc. of the 7th ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages. ACM Press, 1980. P. 83–94.
4. *Miller B. P., Koski D., Lee C. P., Maganty V., Murthy R., Natarajan A., Steidl J.* Nichts dazugelernt — empirische studie zur zuverlässigkeit von unix-utilities. Magazin für professionelle Informationstechnik // iX. 1995. № 9 P. 108–121.
5. *Miller B. P., Fredriksen L., So B.* An empirical study of the reliability of unix utilities // Proc. of the Workshop of Parallel and Distributed Debugging, Digital Equipment Corporation, 1990. P. 1–22.
6. *Jones R., Hosking A., Moss E.* The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 1st edition, 2011.
7. *Appel A. W.* Garbage collection can be faster than stack allocation // Information Processing Letters, Elsevier North-Holland, 1987. № 25(4) P. 275–279.
8. *Tofte M., Talpin J.-P.* Region-based memory management // Information and Computation. 1997. № 132(2). P. 109–176.
9. *Meyer B.* Proving pointer program properties — part 2: The overall object structure // Journal of Object Technology, 2003.
10. *Larson R. G.* Minimizing garbage collection as a function of region size // SIAM Journal on Computing. 1977. Vol. 6, № 4. P. 663–668.
11. *Sinha P.* A memory-efficient doubly linked list, online, version from 18.11.2014, 11 2013. http://www.linuxjournal.com/article/6828.
12. *Parlante N.* Linked list basics, document no. 103 from the stanford computer science education library, 2001. http://cslibrary.stanford.edu/103.
13. *Haberland R.* Tutorials and examples, 11 2016.

14. Jones R, Sun Microsystems Inc. Memory management in the java hotspot virtual machine. Technical Report only from April 2006 http://www.oracle.com/technetwork/articles/java/index-jsp-140228.html.
15. Intl. Organization of Standardization. Iso c++ standard, n4296 from 2014-11-19.
16. *Bornat R.* Proving pointer programs in hoare logic. In Roland Backhouse and José Nuno Oliveira, editors // Proc. of the 5th Intl. Conf. on Mathematics of Program Construction, 2000. Vol. 1 of Lecture Notes in Computer Science, Springer, P. 102–126.
17. *Abadi M., Cardelli L.* A Theory of Objects. New Jersey: Springer, Secaucus, 1996.
18. *Gunter C. A., John C.* (eds.) Mitchell. Theoretical aspects of object-oriented programming — types, semantics, and language design. MIT Press, 1994.
19. *Abadi M., K. Rustan M. Leino.* A logic of object-oriented programs // In Proc. of the 7th Intl. Joint Conf. CAAP/FASE on Theory and Practice of Software Development. Springer, 1997. P. 682–696.
20. *Ehrig H., Rosen B. K.* The mathematics of record handling // SIAM Journal on Computing. 1980. Vol. 9, № 3. P. 441–469.
21. *Abadi M.* Baby modula-3 and a theory of object // Technical Report SRC-RR-95. Systems Research Center, Digital Equipment Corporation. 1993.
22. *K. Rustan M. Leino*. Recursive object types in a logic of object-oriented programs // Nordic Journal of Computing, 1998. Vol. 5, № 4. P. 330–360.
23. *Banerjee A., Naumann D. A., Rosenberg S.* Regional logic for local reasoning about global invariants / In J. Vitek, editor, European conf. on Object-Oriented Programming. Berlin Heidelberg: Springer, 2008. Vol. 5142 of Lecture Notes in Computer Science, P. 387–411.
24. Wikipedia. Region-based memory management. https://en.wikipedia.org/wiki/Region-based_memory_management.
25. *Barnett M., DeLine R., Fähndrich M., K. Rustan M. Leino, Schulte W*. Verification of object-oriented programs with invariants // Journal of Object Technology. 2004. Vol. 3, № 6. P. 27–56.
26. *Müller P.* Modular Specification and Verification of Object-Oriented Programs. PhD thesis // Fern-Universität Hagen, Germany, 2002.
27. D’Souza D. F., Wills A. C. Objects, Components, and Frameworks with UML: The Catalysis Approach. Object Technology Series. Addison-Wesley, 1998.
28. Object Management Group (OMG). Object constraint language version 2.2, 2 2010. http://www.omg.org/spec/OCL/2.2.
29. *Muchnick S.* Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., 2007.
30. *Ramalingam G.* The undecidability of aliasing // ACM Transactions on Programming Languages and Systems, 1994. Vol. 16, № 5. P. 1467–1471.
31. The gnu compiler collection, http://gcc.gnu.org.
32. *Horwitz S., Pfeiffer P., Reps T.* Dependence analysis for pointer variables // In Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York, 1989. P. 28–40.
33. *Khedker U., Sanyal A., Karkare B.* Data Flow Analysis: Theory and Practice. CRC Press, Inc., 1st edition. USA, Florida: Boca Raton, 2009.
34. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 3rd edition. MIT Press, 2009.
35. *Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K.* Efficiently computing static single assignment form and the control dependence graph // ACM Transactions on Programming Lanuages and Systems. 1991. Vol. 13, № 4. P. 451–490.
36. Static single assignment book, latest from july 2014, available online at: http://ssabook.gforge.inria.fr/latest/book.pdf.electronically.
37. *Naeem N. A., Lhoták O.* Efficient alias set analysis using ssa form // In Proc. of the intl. symp. on Memory Management. USA, New York, 2009. P. 79–88.
38. Pavlu V. Shape-based alias analysis — extracting alias sets from shape graphs for comparison of shape analysis precision. Master’s thesis. Austria, Vienna University of Technology, 2010.
39. *Steensgaard B.* Points-to analysis in almost linear time // In Proc. of the 23rd ACM SIGPLAN-SIGACT

symp. on Principles of Programming Languages. USA, New York, 1996. P. 32–41.
40. *Apt K. R.* Ten years of hoare's logic: A survey — part I // ACM Transactions on Programming Languages and Systems, 1981. Vol. 3, № 4. P. 431–483.
41. *Clarke E. M. Jr.* Programming language constructs for which it is impossible to obtain good hoare axiom systems. Journal of the ACM. USA, New York, 1979. Vol. 26, № 1. P. 129–147.
42. *Apt K. R., Olderog E.-R.* Verification of sequential and concurrent programs // SIAM Review. 1993. Vol. 35, № 2. P. 330–331.
43. *Cook S.* Soundness and completeness of an axiom system for program verification // SIAM Journal on Computing, 1978. Vol. 7, № 1. P. 70–90.
44. *Stansifer R.* Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639 // Computer Science Department, Cornell University, 1984.
45. *Meyer B.* Proving pointer program properties — part 1: Context and overview, part 2: The overall object structure. ETH Zürich, Journal of Object Technology, 2003.
46. *Jones N. D., Muchnick S. S.* Even simple programs are hard to analyze // In Proc. of 2nd ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages, 1975. P. 106–118.
47. *Sagiv M., Reps T., Wilhelm R.* Parametric shape analysis via 3-valued logic // ACM Transactions on Programming Languages and Systems. 2002. Vol. 24, № 3. P. 217–298.
48. *Nielson F., Nielson H. R., Hankin C.* Principles of Program Analysis. Berlin, Heidelberg: Springer, 1999.
49. *Hind M.* Pointer analysis: Haven't we solved this problem yet? In USA IBM Watson Research Center, editor // Proc. of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering, 2001. P. 54–61.
50. *Parduhn S. A., Seidel R., Wilhelm R.* Algorithm visualization using concrete and abstract shape graphs // Proc. of the ACM symp. on Software Visualization, 2008. P. 33–36.
51. *Calcagno C., Distefano D., O'Hearn P., Yang H.* Compositional shape analysis by means of bi-abduction // Proc. of the 36th annual ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages, 2009. Vol. 36. P. 289–300.
52. *Suzuki N.* Analysis of pointer rotation. Communications of the ACM, 1982. Vol. 25, № 5. P. 330–335.
53. *Reynolds J. C.* Separation logic: A logic for shared mutable data structures // Proc. of the 17th Annual IEEE symp. on Logic in Computer Science. USA, Washington, DC, 2002. P. 55–74.
54. *Berdine J., Calcagno C., O'Hearn P. W.* Symbolic execution with separation logic // 3rd Asian symp. on Programming Languages and Systems. Japan, Tsukuba, 2005. P. 52–68.
55. *Reynolds J. C.* An Introduction to Separation Logic. 2009. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, course book available online: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr.
56. *Burstall R. M.* Some techniques for proving correctness of programs which alter data structures // Bernard Meltzer and Donald Michie, editors, Machine Intelligence. Vol. 7. Edinburgh University Press, Scotland, 1972. P. 23–50.
57. *Parkinson M., Bierman G.* Separation logic and abstraction // SIGPLAN Notes. 2005. Vol. 40, № 1. P. 247–258/
58. *Restall G.* Introduction to Substructural Logic. Routledge Publishing, 2000.
59. *Hurlin C.* Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. PhD thesis, Université Nice — Sophia Antipolis, France, 2009.
60. *Parkinson M. J.* Local Reasoning for Java. PhD thesis, Cambridge University, England, 2005.
61. *Dodds M.* Graph Transformations and Pointer Structures. PhD thesis. England, University of York, 2008.

Поступила в редакцию 31.07.2018, окончательный вариант — 07.02.2019.

Computer tools in education, 2019
№ 2: 5–**??**
http://cte.eltech.ru
doi:10.32603/2071-2340-2019-2-5-30

Comparative Analysis of Static Methods of Dynamic Memory Verification

Haberland R.[1], postgraduate student, haberland1@mail.ru

[1]Saint-Petersburg Electrotechnical University,
5, building 2, st. Professora Popova, 197376, Saint Petersburg, Russia

Abstract

The article provides an overview of the existing methods of dynamic memory verification; their comparative analysis is carried out; the applicability for solving problems of control, monitoring, and verification of dynamic memory is evaluated. This article is divided into eight sections. The first section is an introduction, the second discusses dynamic memory management problems, the third discusses Hoare's calculus, the fourth considers heap transformations to stack, the fifth introduces the concept of dynamic memory image analysis, the sixth is dedicated to the rotation of pointers, the seventh is on the logic of distributed memory. The last section discusses possible directions of further research in this area; more specifically: recognition at recording level of various instances of objects; automation of proofs; "hot" code, that is, software code that updates itself when the program runs; expanding intuitiveness of proof explanations and others.

Keywords: *dynamic memory verification, Hoare calculus, distributed memory, pointers arithmetics.*

Citation: R. Haberland, "Comparative Analysis of Static Methods of Dynamic Memory Verification," *Computer tools in education*, no. 2, pp. 5–30, 2019 (in Russian); doi: 10.32603/2071-2340-2019-2-5-30

References

1. R. Love, *Linux Kernel Development.* Addison-Wesley Professional, 3rd ed., 2010.
2. C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969; doi: 10.1145/363235.363259
3. W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables," Paul W. Abrahams, R. J. Lipton, and S. R. Bourne, eds., *Proc. of the 7th ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages*, ACM Press, pp. 83–94, 1980; doi: 10.1145/567446.567455
4. B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Nichts dazugelernt — empirische studie zur zuverlässigkeit von unix-utilities," *iX – Magazin für professionelle Informationstechnik*, no. 9, pp. 108–121, 1995.
5. B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *In Proc. of the Workshop of Parallel and Distributed Debugging*, Digital Equipment Corporation, pp. 1–22, 1990.
6. R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Chapman & Hall/CRC, 1st ed., 2011; doi: 10.1201/9781315388021
7. A. W. Appel, "Garbage collection can be faster than stack allocation," *Information Processing Letters*, Elsevier North-Holland, vol. 25, no. 4, pp. 275–279, 1987; doi: 10.1016/0020-0190(87)90175-X
8. M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and Computation*, vol. 132, no. 2, pp. 109–176, 1997; doi: 10.1006/inco.1996.2613

9. B. Meyer, “Proving pointer program properties — part 2: The overall object structure,” *ETH Zürich, Journal of Object Technology*, 2003.
10. R. G. Larson, “Minimizing garbage collection as a function of region size,” *SIAM Journal on Computing*, vol. 6, no. 4, pp. 663–668, 1977; doi: 10.1137/0206047
11. P. Sinha, *A memory-efficient doubly linked list*, [Online], Available: http://www.linuxjournal.com/article/6828
12. N. Parlante, *Linked list basics, document no.103 from the stanford computer science education library*, 2001. [Online], Available: http://cslibrary.stanford.edu/10
13. R. Haberland, *Tutorials and examples*, no. 11, 2016. [Online], Available: https://bitbucket.org/reneH123/
14. R. Jones and Sun Microsystems Inc., “Memory management in the java hotspot virtual machine,” *Technical Report*, April 2006. [Online], Available: http://www.oracle.com/technetwork/articles/java/index-jsp-140228.html
15. Iso c++ standard, n4296, Intl. Organization of Standardization, 19 Nov. 2014. [Online], Available: https://isocpp.org/files/papers/n4296.pdf
16. R. Bornat. “Proving pointer programs in hoare logic,” R. Backhouse and J. N. Oliveira, eds., *Proc. of the 5th Intl. Conf. on Mathematics of Program Construction*, Lecture Notes in Computer Science, Springer, vol. 1, pp. 102–126, 2000.
17. M. Abadi and L. Cardelli, A Theory of Objects, Springer: Secaucus, New Jersey, USA, 1996.
18. C. A. Gunter and John C. (eds.) Mitchell. Theoretical aspects of object-oriented programming — types, semantics, and language design. MIT Press, 1994.
19. M. Abadi, K. Rustan, and M. Leino, “A logic of object-oriented programs,” In *Proc. of the 7th Intl. Joint Conf. CAAP/FASE on Theory and Practice of Software Development*, Springer, pp. 682–696, 1997.
20. H. Ehrig and B. K. Rosen, “The mathematics of record handling,” *SIAM Journal on Computing*, vol. 9, no. 3, pp. 441–469, 1980; doi: 10.1137/0209034
21. M. Abadi, “Baby modula-3 and a theory of object,” *Technical Report SRC-RR-95*, Systems Research Center, Digital Equipment Corporation, 1993.
22. K. Rustan M. Leino. “Recursive object types in a logic of object-oriented programs,” *Nordic Journal of Computing*, vol. 5, no. 4, pp. 330–360, 1998; doi: 10.1007/BFb0053570
23. A. Banerjee, D. A. Naumann, and S. Rosenberg, “Regional logic for local reasoning about global invariants,” J. Vitek, ed., *European conf. on Object-Oriented Programming*, vol. 5142 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, pp. 387–411, 2008; doi: 10.1007/978-3-540-70592-5_17
24. Wikipedia, *Region-based memory management*, [Online], Available: https://en.wikipedia.org/wiki/Region-based_memory_management
25. M. Barnett, R. DeLine, M. Fähndrich, K. Rustan M. Leino, and W. Schulte, “Verification of object-oriented programs with invariants,” *Journal of Object Technology*, vol. 3, no. 6, pp. 27–56, 2004.
26. P. Müller, “Modular Specification and Verification of Object-Oriented Programs. PhD thesis,” *Fern-Universität Hagen, Germany*, 2002.
27. D. F. D’Souza and A. C. Wills, “Objects, Components, and Frameworks with UML: The Catalysis Approach. Object Technology Series,” Addison-Wesley, 1998.
28. Object Management Group (OMG). Object constraint language version 2.2, 2 2010. [Online], Available: http://www.omg.org/spec/OCL/2.2
29. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers Inc., 2007.
30. G. Ramalingam, “The undecidability of aliasing,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1467–1471, 1994; doi: 10.1145/186025.186041
31. The gnu compiler collection, [Online], Available: http://gcc.gnu.org
32. S. Horwitz, P. Pfeiffer, and T. Reps, “Dependence analysis for pointer variables,” In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, pp. 28–40, 1989; doi: 10.1145/74818.74821
33. U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*, 1st ed., Boca Raton, Florida: CRC Press, 2009; doi: 10.1201/9780849332517
34. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
35. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Lanuages and Systems*, vol. 13, no. 4, pp. 451–490, 1991; doi: 10.1145/115372.115320

36. Static single assignment book, latest from july 2014. [Online], Available: http://ssabook.gforge.inria.fr/latest/book.pdf.electronically
37. N. A. Naeem and O. Lhoták. "Efficient alias set analysis using ssa form," In *Proc. Of the intl. symp. on Memory Management*, New York, pp. 79–88, 2009; doi: 10.1145/1542431.1542443
38. V. Pavlu. Shape-based alias analysis — extracting alias sets from shape graphs for comparison of shape analysis precision. Master's thesis, Vienna University of Technology, Austria, 2010.
39. B. Steensgaard. "Points-to analysis in almost linear time," In *Proc. of the 23rd ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages*, New York, pp. 32–41, 1996; doi: 10.1145/237721.237727
40. K. R. Apt, "Ten years of hoare's logic: A survey — part I," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, pp. 431–483, 1981.
41. E. M. Jr. Clarke, "Programming language constructs for which it is impossible to obtain good hoare axiom systems," *Journal of the ACM*, vol. 26, no. 1, pp. 29–147, New York, 1979.
42. K. R. Apt and E.-R. Olderog, "Verification of sequential and concurrent programs," SIAM Review, vol. 35, no. 2, pp. 330–331, 1993.
43. S. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM Journal on Computing*, vol. 1, pp. 70–90, 1978; doi: 10.1137/0207005
44. R. Stansifer, "Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639," *Computer Science Department*, Cornell University, 1984.
45. B. Meyer, "Proving pointer program properties," E*TH Zürich, Journal of Object Technology*, 2003.
46. N. D. Jones and S. S. Muchnick, "Even simple programs are hard to analyze," In *Proc. Of 2nd ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages*, pp. 106–118, 1975; doi: 10.1145/512976.512988
47. M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 3, pp. 217–298, 2002; doi: 10.1145/514188.514190
48. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*, Springer Berlin, Heidelberg, 1999; doi: 10.1007/978-3-662-03811-6_5
49. M. Hind, "Pointer analysis: Haven't we solved this problem yet?," USA IBM Watson Research Center, ed., In *Proc. of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pp. 54–61, ACM, 2001; doi: 10.1145/379605.379665
50. S. A. Parduhn, R. Seidel, and R. Wilhelm. "Algorithm visualization using concrete and abstract shape graphs," In *Proc. of the ACM symp. on Software Visualization*, pp. 33–36, 2008; doi: 10.1145/1409720.1409726
51. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. "Compositional shape analysis by means of bi-abduction," In *Proc. of the 36th annual ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages*, vol. 36, pp. 289–300, 2009; doi: 10.1145/1480881.1480917
52. N. Suzuki, "Analysis of pointer rotation," *Communications of the ACM*, vol. 25, no. 5, pp. 330–335, 1982.
53. J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," In *Proc. Of the 17th Annual IEEE symp. on Logic in Computer Science*, pp. 55–74, Washington, DC, 2002; doi: 10.1109/LICS.2002.1029817
54. J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," In *3rd Asian symp. on Programming Languages and Systems*, pp. 52–68, Tsukuba, Japan, 2005; doi: 10.1007/11575467_5
55. J. C. Reynolds. An Introduction to Separation Logic, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2009. [Online], Available: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr
56. R. M. Burstall, "Some techniques for proving correctness of programs which alter data structures," In *Bernard Meltzer and Donald Michie*, ed., Machine Intelligence, Edinburgh University Press, Scotland, vol. 7, pp. 23–50, 1972.
57. M. Parkinson and G. Bierman, "Separation logic and abstraction," *SIGPLAN Notes*, vol. 40, no. 1, pp. 247–258, 2005; doi: 10.1145/1040305.1040326
58. G. Restall. Introduction to Substructural Logic. Routledge Publishing, 2000.
59. C. Hurlin. Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. PhD thesis, Université Nice — Sophia Antipolis, France, 2009.
60. M. J. Parkinson. Local Reasoning for Java. PhD thesis, Cambridge University, England, 2005.
61. M. Dodds. Graph Transformations and Pointer Structures. PhD thesis, University of York, England, 2008.

Received 31.07.2018, the final version — 07.02.2019.