Министерство образования Российской Федерации
Московский физико-технический институт
(государственный университет)
Факультет управления и прикладной математики
Институт системного программирования

Статическая верификация операций с конечными множествами в программах на языке C

Выпускная квалификационная работа на степень магистра
студента 475 группы Шведа Павла Евгеньевича

ЧЕРНОВИК
Предназначен для слушателей SYRCoSE 2010

Научный руководитель:
к. ф.-м. н. Владимир Рубанов

Москва, 2010

Содержание

1 Введение

1.1 О статической верификации и множествах

Статическая верификация приложений — это анализ исходного когда с целью верифицировать поведение программы без её запуска. В последнее время этот подход к верификации набирает всё большую популярность. Связано это, прежде всего **[citation]**, с увеличением вычислительных мощностей и развитием автоматизированных методов решения вспомогательных математических задач, возникающих при таком анализе.

Появилось множество инструментов и фрэймворков, которые осуществляют такую верификацию: **[citation]**. Особо хотелось бы отметить класс инструментов (**[citation]**), работающий по принципу уточнения абстракции (abstraction refinement). Такие методы основаны на построении математической абстракции программного кода. Процесс верификации начинается с «грубой» абстракции, которая выражена в слишком широких предположениях о том, как будет меняться состояние программы (то есть, значения переменных) при её исполнении. Затем эта абстракция итеративно уточняется, происходит отсечение невозможных путей выполнения, и так продолжается до тех пор, пока не будут отсечены все пути, приводящие к ошибке в программе.

Подобная описанной в предыдущем параграфе техника решает **задачу достижимости** — определения того, может ли программа при своём выполнении, «пройти» из точки входа (как правило, это функция `main()`) в некоторую другую точку, именуемую в задачах верификации «ошибочной». Свойства, верификацию против которых необходиом осуществить, в рамках этого подхода «встраивают» в программный код проверяемой программы проверку соблюдения определённых условий. Несоблюдение этих условий ведёт к переходу в «ошибочную» точку.

Как правило, математическая абстракция строится в предположении, что программа оперирует целыми числами, все операторы работы с которыми выражаются с помощью линейной арифметики. Однако программы используют гораздо большее количество абстракций; они выходят за рамки линейной арифметики, и верификация этих концепций тоже есть богоугодное дело. Например, инструмент BLAST поддерживает верификацию указателей и структур языка C, а также имеет дополнительные модули для верификации списков (если для работы со списком используются функции со специальными сигнатурами).

С другой стороны, свойства, которые необходимо верифицировать, также формулируются, как правило, на более абстрактном уровне, чем это возможно на языке C. Например, свойство корректности освобождения памяти можно сформулировать так: «при освобождении указателя его значение должно принадлежать множеству ранее выделенных указателей».

Задаче верификации корректности работы с динамической памятью столько же лет, сколько самой динамически выделяемой памяти. Также свойства, формулируемые в терминах множеств, можно использовать для верификации корректности блокировок. Можно заметить, что эти свойства в качестве моделей используют только конечные множества, элементы в которые добавляются/удаляются по одному. Для проверки верифицируемых свойств используются функции принадлежности элемента множеству (см. выше) и функция проверки пустоты множества (например, утечка памяти происходит, когда «множество выделенных указателей непусто»). В дальнейшем, мы

будем говорить о конечных множествах, имея в виду только эти операции.

В нашей работе мы предлагаем расширить язык C некоторыми из операций работы с множествами и реализовать механизм их верификации, модифицировав код инструмента BLAST **[citation]**. Мы также проводим сравнение нашей реализации с уже известными подходами к решению проблемы верификации в терминах множеств.

1.2 Верификация программ инструментом “BLAST”

Для сравнения используемых подходов в данной работе мы использовали BLAST **[citation]**. “BLAST” расшифровывается как “Berkely Lazy Abstraction Software verification Tool”. Этот инструмент решает задачу достижимости (проверить, достижима ли определённая строка программы из какой-то точки входа), для чего реализует подход CEGAR (“CounterExample-Guided Abstraction Refinement” — пересмотр абстракции на основе контрпримеров) **[citation]**, используя для поиска предикатов интерполяцию Крэйга в области линейной арифметики и неинтерпретированных функций **[citation]**. Опишем, как происходит верификация, более подробно.

Как уже было указано во введении (1.1), процесс верификации организован как постепенное уточнение грубой абстракции до состояния, когда она способна продемонстрировать недостижимость нужной строки. Впрочем, если программа содержит ошибку, процесс должен закончиться построением контрпримера — последовательност переходов с графе потока управления, соответствующий некоторому набору выходных данных.

Абстракция, используемая в BLAST, — это так называемое “Abstract Reachability Tree” (ART) — абстрактное дерево достижимости. Это дерево, фактически, является деревом путей в программе; его вершины — это строки в программе, а переходы помечены basic block-ами, условными и безусловными переходами, или вызовами функций. В корне находится точка входа в программу. Каждая вершина помечена набором линейных ограничений в пространстве переменных программы. Если этим ограничениям не удовлетворяет ни одна комбинация значений переменных, то значит, строка, помеченная ими, недостижима.

Таким образом, цель процесса верификации — пометить невыполнимыми ограничениями все ошибочные точки. Изначально, в грубой абстракции, все точки помечены ограничением *true*, допускающим любую комбинацию.

Линейные ограничения (называемые предикатами) представляют собой конъюнкции предикатов, полученных при анализе контрпримеров. Такую абстракцию называют Декартовой абстракцией **[citation]**, а эти ограничения — регионами.

Под контрпримером тут понимается следующее: на каждом этапе мы исходим из предположения, что данная абстракция или не находит ошибочных точек (возможно, они просто принципиально недостижимы из точки входа), или же в ней все ошибочные точки помечены невыполнимыми ограничениями. Если это действительно так, то процесс верификации закончен. Если это не так, то имеется *контрпример* к вышеизложенному предположению — путь от корня к допустимой ошибочной точке. Этот путь преобразуется в логическую формулу в форме Single Static Assignment **TODO** , и, если, конечно, эта формула невыполнима (то есть, по этому пути действительно возможна ошибка), путь анализируется с помощью интерполяции Крэйга.

Интерполяция Крэйга для двух формул, конъюнкция которых невыполнима, позволяет заменить одну из них более грубой формулой, достаточной, чтобы сохранить

невыполнимость конъюнкции. Из этих «грубых» интерполянтов и составляются затем регионы.

Конечно, после процедуры нахождения предикатов (“predicate discovery”), необходимо ещё и для каждой строки программы составить из них регионы. Этим занимается процедура анализа постусловий. После каждого перехода по ребру, эта процедура (описанная в статье Graf и Saidi **[citation]**), определяет, будет ли выполняться каждое из линейных ограничений после выполнения блока или условного перехода, которым может быть помечено ребро. Затем определяется, не будут ли эти линейные ограничения невыполнимыми (опять же, с помощью solver-а), и делается вывод, каким будет дочерний регион.

Если в графе потока управления есть цикл, то он способен сгенерировать бесконечное число путей в ART (пройти по циклу один раз, пройти два раза, пройти три раза, и. т. д.). Для предотвращения этого используется механизм покрытия: если в одной из вершин ART регион для данной строчки покрывается регионом другой вершины ART (то есть, ограничения другого региона следуют из ограничений первого), то первая вершина считается покрытой, и переходы из неё не исследуются. Действительно, все варианты поведения программы полностью определяются значением её переменных, и поэтому исследование путей из покрытой вершины не принесёт новой информации, если будет исследован покрывающий.

Чтобы расширить данный подход, внеся в него возможность судить о конечных множествах, необходимо модифицировать процедуры анализа контрпримеров, процедуру поиска постусловий и процедуру проверки покрытия. При этом регионы не обязательно должны выражаться конъюнкцией линейных ограничений.

1.3 Имеющиеся способы верификации множеств

К решению задачи верификации конечных множеств можно подойти несколькими способами.

Во-первых, можно реализовать проверку этого свойства и на языке C (например, с помощщю хэш-таблиц). Такая реализация не полагается на одну лишь линейную арифметику, а сильно использует работу с указателями, методы построения абстракций для которой ещё не так разработаны. Поэтому, предположительно, инструменты верификации не смогут сделать вывод о корректности или некорректности такой программы.

Во-вторых, можно реализовать проверку операций со множествами с помощью расщепления путей (иногда это называют “universal quantificaiton trick”). Грубо говоря, за состояние множества отвечает одна переменная, а каждая операция над множеством будет сопровождаться недетерминированным ветвлением[1]. Образовавшиеся в результате выполнения серии операций трассы выполнения программы будут представлять собой весь спектр возможных состояний множества, и если последовательность опрераций над множествами может привести к ошибочному состоянию, это будет обнаружено тогда и только тогда, когда такая ситуация произошла на одном из путей. Мы опишем это подробней позже.

[1]такое ветвление заведомо не накладывает никаких ограничений на то, какая из веток будет выполняться далее, и инструмент верификации будет вынужден проверить оба пути на корректность.

Эти подходы были известны и ранее. В нашей работе мы предлагаем другой подход. Третий вариант реализации — встроить возможность судить об операциях со множествами в движок инструмента верификации. Для этой цели следует математически описать, как операции со множествами будут влиять на основные механизмы суждения о выражениях в области линейной арифметики, и реализовать это в рамках инструмента верификации.

Математическая абстракция для предложенного подхода была построена с целью устранить известные недостатки первых двух подходов.

Во-первых, устарнена необходимость анализировать сложный программный код, чья структура выходит за пределы области линейной арифметики.

Во-вторых, предотвращается «комбинаторный взрыв» количества путей, подлежащих проверке — то есть, отстутсвуют ветвления в функцийх, реализующих операции со множествами. Это достигается посредством усложнения формул, с которыми работает solver[2], однако, имеющиеся исследования **[citation]** показывают, что такой подход приводит к более быстрой работе.

[2]программа, осуществляющая проверку выполнимости логических формул.

2 Верификация конечных множеств

Как мы уже упомянули, работа со множествами будет осуществляться с помощью специальных функций, которые имеют специальное значение. Пример программы, которая использует эти функции можно посмотреть на рисунке **??**. По названиям этих функций интуитивно понятно, какую операцию со множествами они представляют.

Однако символы, с которыми оперируют эти функции, не являются множествами как таковыми. Точно так же, как выражения и переменные не являются числами (хотя при подстановке и вычислении они дают в результате число), эти символы лишь представляют собой некоторый «паттерн», которому могут удовлетворять разные множества (состоящие из чисел) в зависимости от конкретных значений переменных или ограничений над ними.

Реализация операций над множествами в движке инструмента верификации работает в терминах этих «паттернов», оперируя с символами программы: выражениями, переменными и пр. Программист же, употребляя ту или иную операцию, мыслит в терминах значений, которые будут иметь добавляемые выражения, и множеств, состоящих из этих значений.

Поэтому в данном разделе мы сначала определим, какие именно множества задаются с помощью тех или иных операций, а затем предложим процедуры символьной верификации свойств «паттернов» и докажем, что эти процедуры действительно корректно верифицируют и значения множеств, которые имеются в виду программистом.

2.1 Используемая теория

В этом разделе мы опишем теорию, в терминах которой будет осуществляться работа с множествами. Мы покажем, что она действительно представляет собой суждения о конечных множествах.

Для начала укажем, как мы, с точки зрения этой теории представляем программу. В этом разделе мы полагаем, что мы находимся в конкретной локации программы.ололо Это означает, что переменные с одинаковыми именами имеют одинаковые значения.

Пусть $\mathcal{D}$ — множество возможных значений переменных. Обозначим за $\mathcal{E}(\mathcal{D})$ множество выражений языка C над переменными, которые принимают значения из множества $\mathcal{D}$. Значения переменных в данной локации определяются **абстрактным состоянием программы** π. По определению это — непротиворечивые ограничения в пространстве переменных, которые определяют множество допустимых комбинаций их значений. В случае, если известны точные значения всех переменных (то есть π состоит из одной точки), абстрактное состояние называется **конкретным**. Множество всех значений выражения $e \in \mathcal{E}(\mathcal{D})$, которое оно может принимать при всех принадлежащих π комбинациях переменных, назовём **проекцией** e **на** π и обозначим как π_e.

В теории, с которой оперирует BLAST, $\mathcal{D} = \mathbb{Z}$, а π есть конъюнкция интерполянтов Крэйга (при уточнении абстракции) или конъюнкция постусловий в формуле пути (при получении предикатов), в зависимости от контекста. Под π в нашей теории тоже будут подразумеваться разные вещи в зависимости от контекста.

Мы будем рассматривать каждое множество как историю его создания путём добавления или удаления выражений. При этом, напомним, мы по-прежнему предполагаем, что переменные с одинаковыми именами имеют одинаковые значения. [3]

[3] Для того, чтобы это свойство выполнялось на реальной трассе выполнения программы,

Таблица 1: Функции для работы со множествами

Функция	Значение
Конструирование множеств	
`S = SetEmpty()`	Занести в S пустое множество
`S = SetAdd(T,expr)`	Занести в S объединение множества T и множества из одного значения выражения *expr*
`S = SetDel(F,expr)`	Занести в S разность множества F и множества из одного значения выражения *expr*
Проверка состояния множеств	
`x = SetInTest(S,expr)`	Проверить, есть ли в S значение выражения *expr*, и присвоить x соответствующее булево значение (0 или 1)
`x = SetEmptyTest(S)`	Проверить, пусто ли S, и присвоить x соответствующее булево значение (0 или 1)

Формально определим интуитивно введённый «паттерн»: C — **состав множества над выражениями** $\mathcal{E}(\mathcal{D})$, если это либо

- **пустое множество**: $C = \emptyset$, либо
- **добавление**: $C = C' + x$, где C' — состав над $\mathcal{E}(\mathcal{D})$, а $x \in \mathcal{E}(\mathcal{D})$; либо
- **удаление**: $C = C' - y$, где C' — состав над $\mathcal{E}(\mathcal{D})$, а $y \in \mathcal{E}(\mathcal{D})$.

Теперь можно формально определить, какие именно множества получаются в результате конструирования составов. Множество $c \subseteq \mathcal{D}$ называется **интерпретацией** состава C над $\mathcal{E}(\mathcal{D})$ при заданном абстрактном состоянии $\pi \neq \emptyset$ (запись: $c \in [C]_\pi$), если

$$C = \emptyset \implies c = \emptyset \tag{1}$$

$$C = C' + x \implies \exists a \in \pi_x \, \exists c' \in [C']_{\pi \cap \{x=a\}} \to c = c' \cup \{a\} \tag{2}$$

$$C = C' - y \implies \exists b \in \pi_y \, \exists c' \in [C']_{\pi \cap \{y=b\}} \to c = c' \setminus \{a\} \tag{3}$$

Для $\pi = \emptyset$ составов не существует. Ограничения на π, которые введены в эти формулы, имеют следующий смысл: значения выражений, содержащих общие переменные, должны быть связаны друг с другом в различных «частях» состава.

В программу для конструирования множеств вводится несколько функций. Эти функции приведены в таблице 1. Они соответствуют конструированию состава множества; подстановку переменных (и ограничения) определяют прочие операторы программы.

Теперь покажем, что интерпретации составов, сконструированных в произвольной программе, всегда есть конечные множества. Рассмотрим теперь произвольную

необходимо привести её в форму Static Single Assignment — путём увеличения количества переменных, добиться того, чтобы каждая переменная не менялась после присвоения ей значения

трассу выполнения некоторой программы, использующей операторы, конструирующие состав множеств. Не переставаем при этом продолжать считать, что эта трасса записана в форме Single Static Assignment, то есть переменные не меняют свои значения.

Теорема 1. *Для трассы длины N максимальное число операций добавления и удаления в/из множества есть $\Omega(N)$.*

Доказательство. Действительно, поскольку на вызов неопределённой функции (каковыми и являются функции конструирования состава) в трассе отводится некоторое константное количество шагов α, то максимальное число операций конструирования не превышает $\frac{N}{\alpha}$. Однако программа, в которой не содержится ничего другого, кроме как операции добавления во множество (удаления из множества), будет иметь $\lfloor \frac{N}{\alpha} \rfloor$ операций добавления.

Из этого напрямую следует, что максимальное число операций есть $\frac{N}{\alpha}$. □

Перейдём от составов к множествам, которые получаются после подстановок переменных.

Теорема 2. *В процессе выполнения программы любой состав C, вычисляемый вдоль трассы с помощью операций конструирования, заданных в таблице 1, будет определять только конечные множества.*

Доказательство. Размер множества $c \in [C]_m$, очевидно, может превышать размер множества $c' \in C'$, от которого C является производным, не более, чем на один элемент (при применении операции добавления и при условии, что его значение, выбранное согласно абстрактному состоянию, не принадлежит c'). Несмотря на то, что у этого элемента может быть сколь угодно широкое множество значений, во множестве c всё равно окажется не более чем один дополнительный элемент.

Поскольку в каждый момент времени с начала выполнения программы прошло конечное число шагов, то, согласно теореме 1, в трассе встретилось конечное число операций добавления.

Принимая во внимание, что в начале выполнения программы все составы считаются пустыми, получим, что $|c| < +\infty$. □

Также докажем лемму, которая упростит последующие исследования множеств.

Лемма 1 (о конкретизации)**.** *Для каждой конечной трассы для каждого состава C и для каждого c такого, что $c \in [C]_\pi$ и $|c| < +\infty$, верно*

$$\exists \sigma \ \rightarrow (|\sigma| = 1) \wedge (c \in [C]_\sigma) \tag{4}$$

Иными словами, для каждой интерпретации найдётся конкретное состояние (в котором значения всех переменных известны), что она будет оставаться интерпретацией того же самого состава в этом состоянии.

Доказательство. Действительно, рассмотрим последовательность состояний π^n, которая возникает в силу выполнимости условий существования в формулах (2)–(3) для состава C. Поскольку все эти множества последовательно вложены, и их последовательность конечна (в силу конечности трассы), в последнем, по определению, непустом состоянии последовательности будет какая-то точка σ_0. Эта точка будет принадлежать каждому состоянию π^i. Поэтому $\sigma_0 = \sigma$ доказывает (4). □

Отметим важность условия конечности трассы в формулировке леммы 1. Если оно не выполнено, то такой конкретизации не существует: например бесконечная последовательность состояний $\pi^n = \mathbb{Z} \setminus \rho(0, n)$ не имеет общей точки.

2.2 Проверка ветвлений

Простое конструирование множеств не имеет смысла, если в программе нет условных ветвлений в зависимости от состояния множества (т. е. интерпретации). В предлагаемой теории есть возможность вычислять два вида таких ветвлений: принадлежность элемента множеству (точнее, каждому из множеств, заданных определённым составом) и пустоту множества. Но сперва — подробней о том, что из себя представляет ветвление.

Ветвление — это вершина ГПУ, из которой выходят два ребра, на каждом из которых указано условие. Смысл каждого из условий такой: если в родительской вершине состояние программы π_1, то в дочерней оно не уже, чем $\pi_1 \cap cond$, где $cond$ — это самое условие. Особенно важное значения условия имеют при уточнении абстракции: если $\pi_1 \cap cond = \emptyset$, то процедура уточнения может сделать вывод о недостижимости дочернего состояния (за которым, возможно, через некоторое количество шагов следует ошибочное состояние). [4]

В программном коде осуществляется конструкция составов множеств, однако пользователь мыслит в терминах интерпретаций. При этом данному составу может соответствовать несколько интерпретаций. Поэтому корректно составленное условие должно сужать состояние до пустого только в том случае, когда *ни одна* из интерпретаций состава не удовлетворяет проверяемому условию.

Построим формулы для проверки подобных условий и покажем, что они корректны.

2.2.1 Принадлежность элемента множеству

Помимо состава множества, в проверке принадлежности участвует также выражение, принадлежность одного из значений которого (согласно ограничениям) одной из возможных интерпретаций множеств необходимо проверить.

Теорема 3. *Пусть C — некоторый конечный состав, а π — состояние программы. Тогда для произвольного выражения $e \in \mathcal{E}(\mathcal{D})$*

$$\exists v \in \pi_e \; \exists c \in [C]_{\pi \cap \{e=v\}} \rightarrow v \in c \tag{5}$$

истинно тогда и только тогда, когда

$$SAT(\pi \wedge f(e, C)) \tag{6}$$

где $f(e, C)$ — это

$$C = \emptyset \implies false \tag{7}$$

$$C = C' + x \implies (e = x) \vee f(e, C') \tag{8}$$

$$C = C' - y \implies (e \neq y) \wedge f(e, C') \tag{9}$$

[4] Строго говоря, иногда вывода о недостижимости не делается. В таком случае либо, встретив ошибочное состояние и получив новые предикаты, верификатор уточнит абстракцию достаточно хорошо, либо процедура верификации не завершится вследствие внутренней неточности уточнения.

Формула (5), по сути, и определяет принадлежность какого-либо числа, которое в данном состоянии есть результат вычисления выражения e, множеству, являющемуся в том же состоянии интерпретацией заданного состава.

Доказательство. Сразу отметим, что если ограничения π невыполнимы, то невыполнима формула (6) (вне зависимости от f) и не существует такого v, так как π_e пусто. В дальнейшем будем полагать, что π выполнимы.

Докажем теорему по индукции по числу операций в составе (это можно сделать, поскольку состав конечен).

База индукции: $C = \emptyset$. В этом случае единственной интерпретацией является $c = \emptyset$, и (**??**) всегда ложна; в то же время $f(e, \emptyset) = false$, согласно (7).

Переход: пусть для всех возможных π и конечных C длины не большей n теорема 3 верна. Рассмотрим состав на единицу большей длины, и докажем теорему в каждую сторону отдельно.

- Пусть верна формула (5). В формуле (5) положим существующие c и v равными c_0 и v_0 соответственно. Рассмотрим возможные взаимоотношения между C и составом на единицу меньшей длины C', на основе которого построен C.

  1. $C = C' + x$. Тогда, по определению (2), для некоторых $a \in \pi_x$ и $c'_0 \in [C']_{\pi \cup \{x=a\} \cup \{e=v_0\}}$ верно $c_0 = c'_0 \cup \{a\}$. c_0 содержит значение v_0 в том и только в том случае, когда верно

  $$(a = v_0) \vee (v_0 \in c'_0) \tag{10}$$

  – $(a = v_0)$ верно тогда и только тогда, когда $\pi_e \cap \pi_e \neq \emptyset$, то есть, выполнимо

  $$\pi \wedge (e = x) \tag{11}$$

  – $(v_0 \in c'_0)$, учитывая, что v_0 — произвольное значение из π_e, а c' — произвольная интерпретация $[C']_{\pi \cup \{x=a\} \cup \{e=v_0\}}$, влечёт, по предположению индукции, выполнимость

  $$\pi \wedge f(e, C') \tag{12}$$

  (символьное выражение $x = x$, соответствующее выражению $\{x = a\}$, всегда выполнимо).

  Из выполнимости одной из формул (11) или (17) следует выполнимость их дизъюнкции — то есть, справедливость формулы (8).

  2. $C = C' - y$. В этом случае, по определению (3), для любых $b \in \pi_y$ и $c'_0 \in [C']_{\pi \cup \{y=a\} \cup \{e=v_0\}}$ верно $c_0 = c'_0 \setminus \{b\}$. c_0 содержит значение v_0 в том и только в том случае, когда верно

  $$(b \neq v_0) \wedge (v_0 \in c'_0) \tag{13}$$

  – $(b \neq v_0)$ всегда верно тогда и только тогда, когда $\pi_y \cap \pi_e = \emptyset$, то есть, выполнимо

  $$\pi \wedge (e \neq y) \tag{14}$$

– $(v_0 \in c'_0)$, учитывая, что v_0 — произвольное значение из π_e, а c' — произвольная интерпретация $[C']_{\pi \cup \{y \neq b\} \cup \{e = v_0\}}$, влечёт, по предположению индукции, выполнимость

$$(\pi \wedge (y \neq b)) \wedge f(e, C') \tag{15}$$

Из выполнимости обоих формул (19) и (20) следует выполнимость их конъюнкции. Учитывая, что $(e \neq y)$ влечёт $(y \neq b)$, получаем, что их конъюнкция эквивалентна формуле (9).

- Пусть верна формула (6). Рассмотрим возможные взаимоотношения между C и составом на единицу меньшей длины C', на основе которого построен C.

1. $C = C' + x$. Подставив формулу (2) в (6), получим

$$SAT(\pi \wedge ((e = x) \vee (f(e, C')))) \tag{16}$$
$$SAT((\pi \wedge (e = x)) \vee (\pi \wedge f(e, C'))) \tag{17}$$

Выполнимость дизъюнкции означает выполнимость хотя бы одного дизъюнкта; рассмотрим их по отдельности.

– Пусть выполнимо $\pi \wedge (e = x)$. Тогда $\pi_e \cap \pi_e \neq \emptyset$; возьмём произвольную точку из этого множества и обозначим её как v_0. Тогда при $a = v_0$ в формуле (2), во-первых, найдётся интерпретация c'_0 для состава C' (поскольку множество $\pi \cap \{x = a\}$ не пусто, а во-вторых, v_0 будет принадлежать любой интерпретации c, поскольку $c = c'_0 \cup \{a\}$.
Это означает, что v_0 и $c'_0 \cup \{a\}$ являются примерами для формулы

– $(v_0 \in c'_0)$, учитывая, что v_0 — произвольное значение из π_e, а c' — произвольная интерпретация $[C']_{\pi \cup \{x = a\} \cup \{e = v_0\}}$, влечёт, по предположению индукции, выполнимость

$$\pi \wedge f(e, C') \tag{18}$$

(символьное выражение $x = x$, соответствующее выражению $\{x = a\}$, всегда выполнимо).

Из выполнимости одной из формул (11) или (17) следует выполнимость их дизъюнкции — то есть, справедливость формулы (8).

2. $C = C' - y$. В этом случае, по определению (3), для любых $b \in \pi_y$ и $c'_0 \in [C']_{\pi \cup \{y = a\} \cup \{e = v_0\}}$ верно $c_0 = c'_0 \setminus \{b\}$. c_0 содержит значение v_0 в том и только в том случае, когда верно

$$(b \neq v_0) \wedge (v_0 \in c'_0) \tag{19}$$

– $(b \neq v_0)$ всегда верно тогда и только тогда, когда $\pi_y \cap \pi_e = \emptyset$, то есть, выполнимо

$$\pi \wedge (e \neq y) \tag{20}$$

– $(v_0 \in c'_0)$, учитывая, что v_0 — произвольное значение из π_e, а c' — произвольная интерпретация $[C']_{\pi \cup \{y \neq b\} \cup \{e = v_0\}}$, влечёт, по предположению индукции, выполнимость

$$(\pi \wedge (y \neq b)) \wedge f(e, C') \tag{21}$$

Из выполнимости обоих формул (19) и (20) следует выполнимость их конъюнкции. Учитывая, что $(e \neq y)$ влечёт $(y \neq b)$, получаем, что их конъюнкция эквивалентна формуле (9).

□

Однако ветвление предполагает, что поток может проходить вдоль каждой ветки, если имеющаяся информация о состоянии программы пока не позволяет судить, по какой именно из веток будет проходить исполнение программы. Поэтому необходими также и предикат, который будет ограничивать поток вдоль другой ветки. При этом, как мы сейчас увидим, соответствующее условие не будет простым отрицанием (6).

Теорема 4. *Пусть C — некоторый конечный состав, а π — состояние программы. Тогда для произвольного выражения $e \in \mathcal{E}(\mathcal{D})$*

$$\exists v \in \pi_e \; \exists c \in [C]_{\pi \cap \{e = v\}} \rightarrow v \notin c \tag{22}$$

истинно тогда и только тогда, когда

$$SAT(\pi \wedge \neg f(e, C)) \tag{23}$$

где $f(e, C)$ определяется формулами (7)–(9) теоремы 3.

Доказательство. Аналогично доказательству теоремы 3. □

2.2.2 Проверка пустоты множества

Нестрого пустоту множества, заданного с помощью какого-либо состава, можно сформулировать следующим образом: «множество пусто тогда и только тогда, когда всякий добавленный элемент был из него затем удалён». Запишем это строго.

Теорема 5. *Пусть C — некоторый конечный состав, а π — состояние программы. Тогда утверждение*

$$\exists c \in [C]_\pi \rightarrow c = \emptyset \tag{24}$$

истинно тогда и только тогда, когда

$$SAT(\pi \wedge g(C, \emptyset)) \tag{25}$$

где $g(C, deleted)$ — это

$$C = \emptyset \implies true \tag{26}$$

$$C = C' - y \implies g(C', deleted \cup \{y\}) \tag{27}$$

$$C = C' + x \implies (x \in deleted) \wedge g(C', deleted) \tag{28}$$

Здесь *deleted* — множество элементов, которые были удалены на последующих шагах конструирования состава.

Доказательство. **TODO** — так же как и теорему 3. □

Вторую ветку, необходимую для проверки непустоты множества, мы запишем по-другому.

Теорема 6. *Пусть C — некоторый конечный состав, а π — состояние программы. Тогда утверждение*

$$\exists c \in [C]_\pi \to c \neq \emptyset \tag{29}$$

истинно тогда и только тогда, когда для некоторого не встречающегося в программе символа α истинно

$$SAT(\pi \wedge f(\alpha, C)) \tag{30}$$

где $f(\alpha, C)$ определяется формулами (7)–(9) теоремы 3.

Иными словами, любая интерпретация состава пуста тогда и только тогда, когда невыполнимо утверждение о том, что не связанный никакими ограничениями символ принадлежит этому множеству.

Доказательство. Действительно, (28) можно эквивалентно переписать следующим образом:

$$\exists c \in [C]_\pi \, \exists v \in \mathcal{D} \to v \in c \tag{31}$$

затем можно переставить кванторы местами:

$$\exists v \in \mathcal{D} \, \exists c \in [C]_\pi \to v \in c \tag{32}$$

Поскольку α не встречается нигде в программе, то на него не наложено никаких ограничений, поэтому $\pi_\alpha = \mathcal{D}$. Поэтому, (31) можно записать так:

$$\exists v \in \pi_\alpha \, \exists c \in [C]_{\pi \cap \{\alpha = v\}} \to v \in c \tag{33}$$

Что, согласно теореме 3, равносильно (29). □

Отметим, что подобная формулировка не могла быть использована в теореме 5, поскольку, если есть непустая интерпретация, то выполнимым является как утверждение о том, что ей принадлежит какой-то элемент, так и о том, что какой-то элемент ей не принадлежит (поскольку, по теореме 2, все интерпретации — конечные множества). Поэтому применение формулы (29) в формулировке теоремы 5 невозможно.

2.3 Процедура верификации

О том, как в общих чертах выглядит процедура верификации в рамках инструмента BLAST, уже было сказано в разделе 1.2. В данном разделе мы опишем, как теория, предложенная в разделе 2 будет применена при верификации в рамках данного инструмента.

Если мы хотим модифицировать процедуру верификации, необходимо предложить модификацию следующих структур данных и операций с ними:

1. **Построение формулы пути**: при заданной трассе (последовательности операторов программы от точки входа до какой-то ошибочной локации), построить формулу пути, которая, с учётом операций со множествами, будет невыполнимой тогда и только тогда, когда выполнение программы вдоль этой трассы невозможно — в силу несовместимости постусловий;

2. **Алгоритм получения предикатов**: при данной невыполнимой формуле пути, получить предикаты, которые будут использованы для уточнения абстракции;

3. **Формат регионов**: предложить структуру данных, которая, в дополнение к Декартовой абстракции будет описывать (неточное) состояние программы при уточнении абстракции;

4. **Вычисление сильнешего постусловия**: при данном регионе и данной операции, вычислить наиболее узкий регион, который будет содержать все состояния программы, в которые можно после данной операции перейти из каждого состояния данного региона;

5. **Вычисление покрытия**: при двух данных регионах, выяснить, содержит ли один регион другой.

Эти операции для теории неинтерпретированных фунекций и линейной арифметики уже реализованы в инструменте BLAST. Нам слудет только реализовать дополнения к ним, с помощью которых будет осуществляться верификация конечных множеств — в рамках предложенной нами в разделе 2 теории.

Однако, операцию построения покрытия нам реализовывать не надо. В самом деле, наша теория заведомо на работает для программ, содержащих циклы. Дело в том, что теория сильно зависит от того, что в каждой точке программы каждая переменная имеет только одно значение. Однико же, если множество конструировалось на протяжении всей программы, то добавленное выражение в начале программы и добавленное в него выражение в конце программы могут иметь разную семантику, хоть и одинаковую символьную запись.

Мы подробнее покажем, как именно это ограничение влияет на возможности верификации конечных множеств, в разделе 2.3.1.

Пока что, мы остановимся на том, что необходимо построить формулу пути, получить предикаты, касающиеся множеств, из этой формулы. Также необходимо построить формат для регионов и процедуру уточнения абстракции.

2.3.1 Построение формулы пути

Построение формулы пути и получение предикатов будут рассмотрены в этой снекции. Формула пути строится на основе трассы – то есть пути от корня ART к одной из её вершин. В процессе верификации, как правило, такие трассы строятся только от корня к ошибочным состояниям.

Трасса строится следующим образом. Сначала она преобразуется так, чтобы каждая переменная встречалась в левой части присваиваний не более одного раза (Static single assignment). Переменные, соответствующие составам множеств, также преобразуются в SSA-форму. Затем каждое ребро (а в базовом блоке - каждое присваивание в отдельности) заменяется булевой фоьрмулой, выражающее точное постусловие. После этого формула пути — это конъюнкция всех этих постусловий.

Для множеств постусловия рассматриваются следующим образом. Для операторов конструирования множеств постусловие не ограничено (тождественно истинно). Для операторов проверки же постусловие накладывает ограничения на переменные, добавленные во множество. Поскольку нам известно, какие именно операторы конструирования множества выполняются вдоль каждой трассы, и каждая переменная не меняет своё значение, то состав множества определён *точно*. Поэтому мы можем воспользоваться теоремами раздела 2.2 для выражения постусловий.

В каждом ветвлении (в точке проверки над множествами) точно известен состав проверяемого множества (благодаря SSA-форме переменных-составов). Также точно известно состояние программы π (это есть конъюнкция постусловий, соответствующих операторам трассы до ветвления). Известен точно и оператор на ветке — поэтому известно, какой именно из теорем 3, 4 (при проверке принадлежности) или 5, 6 (при проверке пустоты) следует воспользоваться, чтобы получить постусловие, ограничивающее значения переменных программы, которые необходимы для перехода по данной ветке.

Остаётся только заметить, что условия теорем 3–6, фактически, представляют собой рекурсивные алгоритмы построения символьных выражений λ. Выражение вида $\pi \wedge \lambda$, выполнимость которого равносильна тому, что проверяемое множество, с учётом встреченных в трассе присваиваний, действительно может удовлетворять условию на рассматриваемом ребре. Это и есть то выражение, которое необходимо для корректности построения трассы.

Рассмотрим пример, изображённый на рисунке **??**.

До локации, помеченной как ошибочной (`error()`), от корня имеется две трассы (см. рис. **??**). На том же рисунке указаны формулы пути, полученные из этих трасс. Формулы (**??**) и (**??**) (одинаковые по виду, поскольку у множеств одинаковый состав вдоль этих трасс) получены из условия теоремы 3[5]. Чему при этом равно нужное в условии теоремы π, также указано на этом рисунке.

Хоть формулы (**??**) и (**??**) не отличаются, именно различие в π вносит разницу в то, какие интерпретации будут соответствовать трассам. И действительно, формула пути №1 выполнима, в то время, как формула пути №2 — невыполнима. Поэтому по первому пути невозможно приёти в ошибочное состояние, а по второму — можно.

Это соответствует нашим ожиданиям от интерпретаций соответствующих множеств и выполнимости формул, которые мы строим для проверки этого. С получением предикатов ситуация немного сложнее.

[5]излишняя, на первый взгляд, конъюнкция с $false$ есть следствие применения формулы (7).

Рис. 1: Пример программы, использующей операции со множествами

```
1  int main()
2  {
3          int a,b;  Set S;
4          S = SetAdd(S, 10);
5          a = 10;
6          if (condition)
7                  b = 5;
8          else
9                  b = 10;
10         S = SetDel(S, b);
11         if (SetIn(S,a))
12                 error();
13         return 0;
14 }
```

Рис. 2: Трассы и формулы путей для программы на рисунке **??**

Трасса 1

```
S = SetEmpty();
S1 = SetAdd(S, a);
a = 10;
b = 5;
S2 = SetDel(S1, b);
Assume (SetIn(S2,a))
```

Формула пути 1

$$
\begin{aligned}
true \wedge & \quad (34)\\
a = 10 \wedge & \quad (35)\\
true \wedge & \quad (36)\\
b = 5 \wedge & \quad (37)\\
true \wedge & \quad (38)\\
(b \neq 10) \wedge ((a = 10) \vee false) & \quad (39)
\end{aligned}
$$

$$\pi = \{a = 10\} \cap \{b = 5\}$$

Трасса 2

```
S = SetEmpty();
S1 = SetAdd(S, a);
a = 10;
b = 10;
S2 = SetDel(S1, b);
Assume (SetIn(S2,a))
```

Формула пути 2

$$
\begin{aligned}
true \wedge & \quad (40)\\
a = 10 \wedge & \quad (41)\\
true \wedge & \quad (42)\\
b = 10 \wedge & \quad (43)\\
true \wedge & \quad (44)\\
(b \neq 10) \wedge ((a = 10) \vee false) & \quad (45)
\end{aligned}
$$

$$\pi = \{a = 10\} \cap \{b = 10\}$$

Рис. 3: Пример программы, демонстрирующей сложности с получением предикатов

```
1  int main()
2  {
3          int a,b;  Set S;
4          b=0;
5          a=1;
6          S = SetAdd(S, a);
7          a=2;
8          b=1;
9          S = SetDel(S, b);
10         if (! SetEmpty(S,a))
11                 error();
12         return 0;
13 }
```

Дело в том, что для получения корректных предикатов в каждой точке программы необходимо, чтобы SSA-преобразование удослетворяло определённому условию. Это условие можно сформулировать так: пусть SSA-образ переменной x (отличающийся в разных точках трассы) записывается в виде x_i, где $i \in \mathbb{N}$. Тогда, если до присваивания x-у нового значения образ переменной x был x_k, то в трассе *после присваивания* x_k никогда не будет образом переменной x.

Действительно, рассмотрим программу, изображённую на рис. **??**. Трасса, ведущая к ошибке, и соответствующая формула пути представлены на рис. **??**. Формула пути, как видно, невыполнима (то есть представленная на рис. **??** программа корректна).

Однако рассмотрим результат процедуры получения предикатов. Интерполяция Крэйга, возможно, даст нам формулы такого вида: в разрезе между конъюнктов (**??**) и (**??**) — формулу $a_0 = 1$, а в разрезе между конъюнктов (**??**) и (**??**) — формулу $b_1 = 1$. Поскольку переменные a_0 и b_1 получились в результате SSA-преобразования, полученные предикаты будут иметь вид $a = 1$ и $b = 1$.

Но можно заметить, что при дальнейшем анализе трассы $a = 1$ и $b = 1$ *не будут выполняться одновременно в точке ветвления (строке 10)*! Более того, $a \neq b$ *в любой точке программы*. Причиной этого является то, что переменная a_0 оказалась в трассе уже после того, как было присвоено новое значение новому образу переменной a — a_1.

Возможно, получив изначальные интерполянты Крэйга ($a_0 = 1$ и $b_1 = 1$), можно было каким-либо образом представить предикаты, выраженные не в терминах простых целочисленных переменных исходной программы, а в теории, использующей множества. Иными словами, можно было бы не игнорировать информацию о том, что в анализируемой формуле какие-то части были получены для проверки отношений на множествах. Однако в данной работе не рассматривались методы, требующие изменения процедуры получения предикатов.

Таким образом, предложенный способ построения предикатов будет корректно работать только если каждая операция конструирования множествадобавляет/удаляет

Рис. 4: Трасса и формула пути для программы на рисунке **??**

Трасса

```
S0 = SetEmpty();
a0 = 1;
b0 = 0;
S1 = SetAdd(S0, a0);
a1 = 2;
b1 = 1;
S2 = SetDel(S1, b1);
Assume (! SetEmpty(S2));
```

Формула пути

$$
\begin{aligned}
true \wedge & \quad (46)\\
a_0 = 1 \wedge & \quad (47)\\
b_0 = 0 \wedge & \quad (48)\\
true \wedge & \quad (49)\\
a_1 = 2 \wedge & \quad (50)\\
b_1 = 1 \wedge & \quad (51)\\
true \wedge & \quad (52)\\
\neg(a_0 = b_1) & \quad (53)
\end{aligned}
$$

$$\pi = \{a_0 = 1\} \cap \{b_0 = 0\} \cap \{a_1 = 2\} \cap \{b_1 = 1\}$$

в/из него выражение, которое содержит только не встречающиеся в оставшейся части программы переменные, и при этом программа не содержит операции со множествами в теле циклов.

Вышеописанное представляет собой серьёзный недостаток предлагаемой теории, который значительно ограничивает её применимость.

2.3.2 Построение постусловий

Построение сильнейших постусловий — это процедура получения по региону и операции нового региона, который содержит все возможные состояния программы, в которые эта операция переводит программу из какого-либо состояния исходного региона. Это можно записать как $A' = post(A, op)$.

Разумеется, для того, чтобы строить полезные регионы для операций со множествами, имеющуюся в BLAST декартову абстракцию $\mathcal{C}$ необходимо расширить. Мы расширим её с помощью декартова произведения с абстракцией $\mathcal{S}$ для множеств — то есть, $A = (\mathcal{C}, \mathcal{S})$.

Итак, определим, какие задачи будут у той части описания региона $\mathcal{S}$, которая отвечает за множества. По информации, хранящейся в этом регионе, процедура построения постусловия должна

- **для операций конструирования** — сохранить информацию о составах множеств и об ограничениях на выражения
- **для операций проверки** — проверить, выполнимо ли ограничение при данной информации о состояни A, и, если нет, установить $A' = \emptyset$, а если да, то $A' = A$.

Согласно сказанному в секции 2.3.1, процедура получения предикатов выявляет лишь дополнительные линейные ограничения в пространстве переменных. Эти ограничения добавляются в регион на этапе присваивания этим переменным их значений, а не на этапе построения состава множества. Поэтому для операций конструированя не нужно каким-либо образом изменять регион $\mathcal{C}$.

Чтобы можно быть проверить выполнимость операции проверки принадлежности/пустоты множества, необходимо иметь информацию о его составе, и об ограничениях, наложенных на переменные, выражения над которыми были добавлены во множества, находящиеся в текущей локации. При этом ограничений, которые хранятся в декартовой абстракции $\mathcal{C}$, недостаточно. Действительно, с момента добавления во множество переменная могла изменить своё значение, и те предположения, на основе которых можно судить о невыполнимости проверки множества, уже не будут справедливы в данной точке абстракции.

Пример уже был приведён: если рассмотреть программу, изображённую на рис. **??**, и предположить, что анализ контрпримера (трассу можно увидеть на рис. **??**) вернул предикаты $a = 1$ и $b = 1$, то в точке абстракции, соответствующей строке 11, они уже не будет справедливы. Однако, если бы они были «запомнены» на соответствующих строчках при конструигровании множества (6-й и 9-й соответственно), то при пересмотре абстракции для ветвления в строке 10 информации для отсечения ошибочного пути было бы достаточно.

Для того, чтобы сохранить информацию о переменных, добавляемых во множество, регион $\mathcal{S}$ содержат следующую информацию:

- **Таблица внутренних переменных** $\mathcal{S}.map$ — таблично заданная функция; записи вида (x, x_i) указывают, что при добавлении во множества выражения, в котором участвует переменная x, она будет заменена на x_i. Эта информация обновляется при каждом присваивании значения переменной x. Таким образом, строится некая таблица, представляющая переменнае в виде “Static Single Assignemnt”.

- **Множество внутренних предикатов** $\mathcal{S}.preds$ — множество отношений, которым удовлетворяют внутренние переменные. Эти отношения выичсляются с помощью замены переменных согласно $\mathcal{S}.map$ во всех предикатах абстракции $\mathcal{C}$, и новые отношения добавляются на каждом шаге уточнения абстракции. Таким образом, эти предикаты, в отличие от $\mathcal{C}$, предоставляют информацию и о тех ограничениях, которым удовлетворяли переменные на предыдущих шагах уточнения абстракции.

- **Информацию о составах множеств** $\mathcal{S}.sets$ — действительно, все переменные, являющиеся множествами, известны. Также, поскольку операции со множествами не встречаются в теле циклов, для каждой трассы, соответствующей пути в ART от корня к данной вершине, составы всех множест будут идентичны. Поэтому информация о составе в следующем регионе, которая тривиально вычисляется на основе состава в предыдущем регионе и операции конструирования, будет точной. Разумеется, при этом следует заменять переменные в выражениях согласно $\mathcal{S}.map$.

При вычислении постусловия операции проверки множества, в условиях теорем

Рис. 5: Пример программы, демонстрирующей необходимость

Исходный код

```
int main(int x, int z)
{
  Set S;
  S = SetAdd(S, z);
  if (SetInTest(S,x))
    if (x <> z)
      error();
}
```

Трасса до `error()`

```
S0 = SetEmpty();
S1 = SetAdd(S0, z);
Assume (SetInTest(S1,x));
Assume (x <> z);
```

Формула пути:

$$x = z \wedge \tag{54}$$
$$x \neq z \tag{55}$$

3, 4 (при проверке принадлежности) или 5, 6 (при проверке пустоты), записанных для $\mathcal{S}.sets$, за ограничения принимаются $\mathcal{S}.preds$.

Это вычисление постусловий не всегда достаточно, посольку иногда требуется введение нового предиката в ограничения $\mathcal{C}$ при переходе по ребру, соответствующему процедуре проверки множества. Например, на рис. **??** изображена программа, для которой процедура интерполяции выдаст предикат $x = z$, но он не будет добавлен в $\mathcal{S}$ при переходе по ребру, соответствующему `if SetInTest()`, поскольку он должен быть сперва добавлен в $\mathcal{C}$, но для этого нет возможности. В результате инструмент верификации сочтёт эту корректную программу некорректной. Однако, эксперименты (см. раздел **??** показывают, что в стандартных случаях применения множеств эта процедура вычисляет постусловия с достаточной точностью.

Отметим также, что предложенный в этом разделе способ сохранения предикатов с помощью замены в них переменных не всегда позволяет устранить основной недостаток предложенной теории, о котором говорилось в конце раздела 2.3.1. Для трассы, изображённой на рис. **??**, процедура поиска предикатов могла вернуть предикат $a = b$, который, повторимся, ни в одной точке программы не был бы верен, и, соответственно, не был бы запомнен в соответствующем множестве региона $\mathcal{S}$.

3 Оценка корректности и производительности

В качестве способа оценить применимость предлагаемого в данной работе алгоритма и сравнить её с имеющимися решениямы была выбрана верификация корректности работы с памятью. Верификация модели памяти языка C была реализована тремя различными способами, а затем были проведены эксперименты, измеряющие корректность и производительность предложенных алгоритмов.

В этом разделе подробно описаны способы реализации операций проверки корректности операций работы с памятью всеми рассматриваемыми методами. Также мы предлагаем описание реализации предложенного в разделе 2 данной статьи подхода к верификации конечных множеств в рамках инструмента BLAST. С него и начнём.

3.1 Как предлагаемая теория встроена в BLAST

Чтобы пользовательские программы могли использовать операции со множествами, создан специальный заголовочный файл, в котором описан тип множества (простой `int`; информация о типе не используется в алгоритме проверки инструмента “BLSAT”) и объявлены функции, представленные в таблице 1. Чтобы позволить множествам хранить любые типы данных, в этих функциях используются “variadic arguments”, т. е. неопределённое число аргументов, что позволяет не указывать тип.

Несмотря на то, что у этих функций отсутствуют определения (тела), ГПУ вызывающих процедур будет сгенерирован точно таким же образом, поскольку для каждой функции генерируется отдельный граф. Также на каждый выхов функции будет приходиться отдельное ребро, поэтому и на каждую операцию со множествами также будет приходиться отдельное ребро в ГПУ.

Несмотря на то, что в данной работе мы описывали ветвления как два ребра с различными условиями на каждом, в действительности же ветвление выглядит по-другому. Как можно увидеть из таблицы 1, при прохождении вдоль операций проверки, результат сначала записывается в отдельную переменную (назовём её `x`), а затем на рёбрах ветвления проверяется равенство нулю её значения. Поэтому в формуле пути условие будет записано следующим образом:

$$(x = 1 \wedge pred_1) \vee (x = 0 \wedge pred_2)$$

где $pred_1$ и $pred_2$ — символьные выражения, о которых мы говорили, что они записываются на рёбрах соответствующего ветвления.

Алгоритмы построения символьных формул, фактически, описаны в теоремах раздела 2.2; их переложение на функциональный язык программирования OCaml не представляет ни сложности, ни интереса. Алгоритмы построения регионов также не отличаются от предложенных в разделе 2.3.1. Для манипуляций с сущностями программы (такими как «переменные», «выражения»), а также для проведения интерполяции, была переиспользована существующая инфраструктура, реализованная в инструменте “BLAST”.

3.2 Детали реализации сравниваемых подходов

В этом разделе мы (подробней, чем в 1.3) описываем известные решения задачи верификации множеств: хранение множества в хэш-таблице и верификация её с

помощью неспециализированных средств анализа; а также использование расщепления путей. Мы приводим нестрого обоснованные предположения о преимуществах и недостатках того или иного метода, и подкрепляем их ссылками на эксперименты.

3.2.1 Верификация операций с памятью с использованием множеств

3.2.2 Хэш-таблицы

Хэш-таблицы являются одним из стандартных способов реализации концепции множества. Действительно, основные операции хэш-таблицы — это добавление в неё элементов и быстрая (за амортизированное время $O(1)$) проверка, был ли элемент когда-либо доавлен в неё **[citation]**. Хэш-таблицы присутвтуют как библиотечная стурктура данных во многих языках программирования. В данной работе мы используем хэш-таблицу с закрытой адресацией (то есть для совпадающих ключей значения добавляются в связный список).

Тем не менее, хэш-таблица — достаточно сложная структура данных для того, чтобы быть качественно проверенной с помощью инструментов статического анализа. Наиболее «тонкое» место в её струкутре — наличие массива и списка, сложных контейнеров, в которые помещаются добавляемые элементы. Описать с помощью линейных ограничений место в этом контейнере, в которое попадает элемент, затруднительно. Поэтому анализ, предположительно, будет давать некорректные результаты.

Впрочем, гораздо лучше обтекаемых формулировок, о сложности верификации хэш-таблиц свидетельствует эксперимент, результаты которого представлены в таблице **??**. Как видно, использование этой структуры данных не позволяет инструменту отличить корректные программы от некорректных.

3.2.3 Расщепление путей

Идея верификаций операций с выделением и освобождением памяти приписывается разработчикам инструмента “Bandera” **[citation]**. Однако, в этой работе не приводится конкретного алгоритма, соответствующего данному методу. Восполним же этот пробел.

Сперва хотелось бы вновь отметить, что «расщепление путей» — это не конкретный алгоритм, а подход к построению алгоритмов, опирающихся на множества. Ограничения этого подхода не позволяют выразить точную верификацию всего спектра операций с конечными множествами. Поэтому в данной работе мы выбрали для сравнения ограниченный спектр операций: отсутствие повторного освобождения памяти и отсутствие утечек (т. е. проверка того, что вся выделенная память освобождена); при этом не проверенным осталось свойство «указатель, переданный в функцию освобождения, указывает на выделенную ранее память».

Важным понятием является неоднократно ранее использованное «расщепление путей». Назовём **расщеплением путей** такое ветвление, для которого при любых ограничениях, переход по каждой из веток не ведёт к недопустимому состоянию. В BLAST для этого обычно используется конструкция `if (undef_func())`, где `undef_func()` — функция, не исеющая определения (тела). Её результатом будет произвольное целочисленное значение, и определить, будет ли оно равно нулю, невозможно. Поэтому неизбежно придётся перейти по обоим веткам этого условного оператора, и в дальнейшем отслеживать в ART два независимых поддерева.

Рис. 6: Модель операций с памятью с использованием расщепления путей

```
 1 int maybe(); //returns an arbitrary bool
 2 #define EPS 0
 3 int counter = 1;
 4 /* One of the pointers in set */
 5 void* M = EPS;
 6 /* One of the pointers deleted from set */
 7 void* F = EPS;
 8 ptr malloc()
 9 { counter += 1;
10   if (maybe()) M = counter;
11         // Intentionally commented out - see text
12   // if (F == counter) F = EPS;
13   return counter; }
14 void free(void* p)
15 { if (M == p) M = EPS;
16   if (F == p) error();
17   if (maybe()) F = p; }
18 void check_leaks()
19 { if (M != EPS) error(); }
```

Идея подхода заключается в том, чтобы отслеживать для проверяемого свойства значение какой-либо характеристической переменной. При этом, при операциях, чьими аналогами были бы операции конструирования множества, пути расщепляются, и характеристическая переменная обновляется только в одном из них. При этом в поддеревьях ART возникают все возможные варианты характеристических переменных.

Вычисление аналогов операций проверки происходит только по характеристическим переменным; при этом если хоть в одном поддереве допускается переход в ошибочное состояние после такой проверки, то это означает, что в программе содержится ошибка. Если же в каждом поддереве такой ошибки нет, то значит, и множество в целом удовлетворяет проверяемому свойству.

Опишем алгоритм проверки вышеуказанных свойств более подробно. Модельные версии процедур работы с памятью можно видеть на рисунке **??**. Здесь `malloc()` и `free(void*)` — стандартные процедуры работы с памятью, а `check_leaks()` — функция, которая вызывается в конце программы и проверяет, не образовалось ли утечек.

Мы также будем использовать специальное значение ε (в исходном коде оно обозначено как `EPS`). При этом $\varepsilon \notin \mathcal{D}$, т. е., это математическая абстракция: значение, не равное никакому другому, встречающемуся в программе. В предлагаемом алгоритме в качестве ε используется значение 0; при этом характеристические переменные не принимают это значение.

Свойства памяти проверяются следующим образом:

- **Проверка отсутствия «утечек памяти»**. За это отвечает характеристическая

переменная M (от "memory"). Изначально она равна ε, а в процессе работы программы поддерживается инвариант *если $M \neq \varepsilon$, то в ней хранится один из указателей на выделенную память*. Для этого на строчке 10 осуществляется расщепление путей, и в одной из ветвей присваивается $M \leftarrow counter$, где $counter$ — указатель на выделяемую память. Очевидно, что инвариант сохраняется при такой операции.

Модельные значения указателя на выделяемую память ($counter$) увеличиваются, чтобы гарантировать, что участки выделенной памяти не перекрываются.

При каждом вызове `free()`, освобождается память по указателю p, и поэтому должна обновиться M. Для поддержки значения инварианта нужно убедиться, что $M \neq p$; это достигается условием и присваиванием на строке 15.

Проверка утечек осуществляется в процедуре `check_leaks()`. Поскольку инвариант верен при входе в неё, наличие в M значения, отличного от ε, означает, что осталась выделенная, но не освобождённая память; на строке 19 имеется переход в ошибочное состояние при выполнении этого условия.

- **Проверка отсутствия «двойных удалений»**. За это отвечает характеристическая переменная F. Изначально она равна ε, а в процессе работы программы поддерживается инвариант *если $F \neq \varepsilon$, то в ней хранится одно из указателей на ранее освобождённую память*. Для этого на строчке 17 осуществляется расщепление путей и в одной из ветвей присваивается $F \leftarrow p$, где p — освобождаемый указатель. Очевидно, что инвариант сохраняется при такой операции.

  При каждом вызове `free()`, до вышеописанного обновления, осуществляется проверка $F = p$. Если это верно, то освобождаемый указатель p равен, согласно инварианту, одному из ранее освобождённых значений. А поскольку все модельные значения выделяемых указателей не равны попарно[6], это действительно будет означать повторное освобождение памяти.

  Осталось лишь заметить, что если какое-то значение p_0 было освобождено дважды, то в момент повторного вызова функции `free()`, в одном из путей $F = p_0$.

3.3 Тестовые данные

Здесь я кратенько описываю, какие программы были выбраны в качестве тестовых, привожу исходные коды, чтобы раздуть объём.

Для оценки производительности сравниваемых подходов автоматически были сгенерированы модельные, «искуственные» программы. Эти программы состоят из последовательного выделения нескольких участков памяти и последовательного их освобождения, за чем, возможно, следовала проверка на наличие утечек памяти. Варьируемыми параметрами были:

- количество выделяемых блоков памяти

[6] если это условие не считать заведомо истинным, то необходимо раскомментировать строку 12. Однако мы предпочли закомментировать её, так как в текущей модели она не необходима для того, чтобы верификация была корректной, а лишь увеличивает время проверки

Таблица 2: Сравнение корректности работы алгоритмов

Алгоритм	Корректная программа	Внесённая ошибка	
		Двойное освобождение	Утечка памяти
Образец	Корректно	Некорректно	Некорректно
Хэш-таблицы	**Некорректно**	Некорректно	Некорректно
Расщепление путей	Корректно	Некорректно	Некорректно
Конечные множества	Корректно	Некорректно	Некорректно

- наличие проверки утечек памяти (как сказано в разделе **TODO** , возможно, это сильно повлияет на алгоритм со множествами)

Для различных вариаций этих параметорв было измерено время работы алгоритма (пользовательское); оно и является параметром, по которому алгоритмы сравниваются. Стоит отметить, что, вследствие особенностей подхода CEGAR, корректная программа верифицируется дольше, чем некорректная. Учитывая также и то, что программ, корректно работающих с памятью больше, чем программ, работающих с ней некорректно, разумно принять время верификации корректной программы как характеристический параметр, по которому сравниваются алгоритмы.

Также в указанные выше программы были внесены ошибки вида «двойное освобождение памяти» и «утечка памяти» с целью проверить корректность предложенных алгоритмов верификации. Время в этом случае не замерялось.

Проверка на программах, взятых из производственного кода, не проводилась. О причинах этого мы расскажем в разделе **??**.

Эксперименты проводились на компьютере с ОС openSuSE 11.2, процессором Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz[7], лимит памяти был установлен в 2 Гб, лимит времени — в 2000 секунд на программу.

3.4 Результаты экспериментов и их анализ

В таблицах **??** и **??** приведены данные о сравнении корректности алгоритмов и времени их работы.

Из таблицы **??** ясно, что, в соответствии с предположениями, сделанными в разделе **??**, решение проблемы выделения/освобождения памяти с помощью хэш-таблиц не обладает достаточной корректностью. Поэтому в дальнейших экспериментах этот подход не участвовал. Решения же с помощью расщепления путей и предложенного алгоритма с конечными множествами справились с внесёнными ошибками отлично.

Таблица **??** позволяет сделать следующие выводы:

- Предложенный метод, в рамках выбранного лимита времени, способен проверить чуть больше модельных программ, чем метод с расщеплением путей. Учитывая, что метода расщепления путей не позволяет проверить всю полноту ошибок, которые можно сделать при работе с памятью, можно полагать, что предлагаемый подход обеспечивает большую корректность работы.

[7] Тем не менее, использовалось только одно ядро

Таблица 3: Сравнение времени работы рассматриваемых алгоритмов

Утечки памяти	Не проверялись		Проверялись	
Алгоритм	Расщепление	Множества	Расщепление	Множества
1 блок	1	1	1	1
2 блока	4	3	5	4
3 блока	41	6	52	10
4 блока	443	17	540	X
5 блоков	1289	36	1553	80
6 блоков	> 2000	70	> 2000	X
7 блоков	> 2000	200	> 2000	X
8 блоков	> 2000	333	> 2000	X
9 блоков	> 2000	X	> 2000	X
10 блоков	> 2000	X	> 2000	X
15 блоков	> 2000	X	> 2000	X

Время указано в секундах. X означает «ошибка CSIsat».

- Предложенный алгоритм, тем не менее, имеет свои ограничения. Как уже было сказано в секции **TODO** , символьные выражения, которые необходимо подвергнуть анализу, более сложны, чем в методе расщепления путей. Поэтому неудивительно, что программа построения интерполянтов Крэйга (“CSIsat”) перестала справляться со столь сложными формулами и стала завершаться аварийно при превышении некоторого соответствующего данному классу задач порога.

3.5 Дальнейшая работа

Несмотря на то, что оптимизации производительности разработанного прототипа не было уделено много внимания, он, всё же, работает быстрее, чем подход с расщеплением путей. Однако аварийное завершение (описанное в предыдущем пункте) возникает не по причине недостаточной оптимизации. Поэтому дальнейшая работа над прототипом (увеличение гибкости работы с множествами, уменьшение количества информации, хранящейся в регионах), при отсуттсивии качественно новых идей или спечиальной оптимизации процедуры получения интерполянтов, не приведёт к улучшению качества инструмента.

Качественно улучшить алгоритм можно с помощью надстройки над LA+EUF интерпояцией, которая позволит выделить операции работы со множествами и проанализировать их отдельно. Важность этого особенно проясняется, если вспомнить об описанном в разделе 2.3.1 серьёзном недостатке предлогаемого подхода, выраженном в том, что в результате интерполяции информация о множествах оказывается потерянной.

Получение интерполятнов, выраженных в терминах, имеющих отношения ко множествам, также помогут разрешить проблемы, связанные с невозможностью использовать метод конечных множеств в программах, содержащих циклы.

Поэтому именно в направлении улучшения процедуры интерполяции, мы считаем,

и целесообразно проводить дальнейшие исследования.

4 Заключение

В данной работы мы показали, что некоторые достаточно распространённые свойства программ, которые проверяются с помощью методов статического анализа, можно сформулировать в терминах операций с конечными множествами. Мы показали, что существующие подходы к этой проблеме не решают её достаточно быстро и корректно.

Мы предложили новый подход к верификации программ, использующих операции с конечными множествами. В разделе 2.1 были представлены примитивы операций со множествами, были предложены алгоритмы построения символьных формул для выражения суждений о построенных с помощью этих операций множеств; была доказана и корректность этих формул. Также были установлены ограничения применимости предложенной теории: операции со множествами не могут находиться в теле циклов.

В разделе 2.3 показано, каким образом верификацию конечных множеств с помощью предложенной теории можно встроить в алгоритм CEGAR с декартововй абстракцией (т. е. в алгоритм, исопльзуемый в инструменте BLAST).

Был разработан прототип реализации предложенного в данной работе подхода и проведены сравнительные испытания его с другими решениями. В качестве платформы для проведения экспериментов были выбран инструмент BLAST и специально сгенерированы программы на языке C. Экспериментальные данные свидетельствуют, что новый алгоритм работает корректно и способен верифицировать более сложные программы, чем имеющиеся решения. Однако, и этот алгоритм наталкивается на ограничения, связанные с избыточной сложностью формул, которые необходимо подвергнуть интерполяции.

Таким образом, предложенный подход показывает себя в экспериментах недостаточно хорошо. Учитывая значительные ограничения в его применимости, можно считать нецелесообразным улучшение применяемых алгоритмов и дальнейшее развитие разработанного прототипа для использования в промышленных системах верификации.