Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Мандрыкин Михаил Усамович

Моделирование памяти Си-программ для инструментов статической верификации на основе SMT-решателей

05.13.11 – математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
д. ф.-м. н., проф.
Петренко Александр Константинович

Москва – 2016

Оглавление

Введение

Актуальность темы

Язык программирования Си продолжает оставаться одним из наиболее широко используемых языков программирования в области системного программирования, в частности при написании операционных систем, драйверов, сред окружения и средств поддержки времени выполнения для различных языков программирования, а так же при написании других систем, предъявляющих высокие требования к производительности или объему исполняемых программ. Так как ко многим из таких систем предъявляются требования не только высокой производительности, но и высокой надежности, а в некоторых случаях и безопасности, продолжает оставаться актуальной задача верификации Си-программ. Эта задача может решаться с применением методов динамической, статико-динамической и статической верификации.

Методы статической верификации отличаются меньшими требованиями к окружению верифицируемой программы, часто требуя для анализа только соответствующий исходный код, а также показывают лучшие по сравнению с методами динамической и статико-динамической верификации результаты при использовании для обнаружения некоторых классов ошибок. Эти качества делают методы статической верификации перспективными, в частности, для верификации ядер операционных систем [1], что в свою очередь является важным аргументом, подтверждающим актуальность развития этих методов.

Область статической верификации программ в настоящее время активно развивается по всем основным направлениям, а именно развиваются автоматическая статическая, дедуктивная верификация и использование языков программирования, интегрирующих программы и соответствующие доказательства корректности. Происходит как развитие и рост практического применения традиционных методов автоматической статической верифика-

ции, так и появление принципиально новых подходов в этой области. Методы дедуктивной верификации развиваются в направлениях интеграции автоматических и ручных подходов к доказательству корректности программ, использованию преимуществ применения специализированных систем типов и автоматизации проверки свойств программ, работающих с динамическими структурами данных. Разрабатываются новые языки, интегрирующие программы и спецификации.

При этом в большинстве современных инструментов статической верификации используются SMT-решатели, и, таким образом, семантика языка программирования, на котором написана верифицируемая программа, так или иначе моделируется с помощью логических формул в теориях (SMT-формул). В зависимости от используемого языка программирования (например, императивного, объектно-ориентированного или функционального, сильно или слабо, статически или динамически типизированого) и языка спецификации (в частности, основанного на логике первого порядка, высших порядков или, например, логике разделения), а также от используемого метода верификации (автоматическая статическая или дедуктивная верификация), подходы к соответствующему эффективному моделированию семантики в виде SMT-формул могут очень существенно различаться как друг от друга, так и от подходов к моделированию семантики тех же языков программирования и спецификации с целью их формального описания на естественном языке (например, в соответствующей документации, стандартах, руководствах и т. д.). Даже небольшие изменения в одном методе моделирования семантики некоторых фиксированных языков программирования и спецификации (например, Си и ACSL) могут приводить к изменению, к примеру, времени работы SMT-решателей на результирующих формулах (и, как следствие, времени работы всего инструмента верификации) в несколько десятков раз. Кроме этого, методы моделирования семантики языков программирования

различаются не только по производительности SMT-решателей на результирующих формулах, но и по ряду специфических свойств, которые могут быть в различной степени важны при использовании соответствующего метода в том или ином инструменте статической верификации. В таком контексте разработка соответствующих методов эффективного моделирования семантики используемых языков программирования и спецификации с помощью SMT-формул становится важной и актуальной задачей.

Для языка Си основной проблемой при моделировании семантики в виде логических формул является моделирование семантики операций с указателями, в частности, указателями на динамически выделяемые области памяти наперед не ограниченного размера, а также моделирование различных приведений типов указателей и случаев использования объединений. Это делает актуальной задачу разработки соответствующих методов эффективного и точного моделирования памяти Си-программ для используемых на практике инструментов статической верификации.

Помимо развития собственно методов моделирования памяти Си-программ важной задачей является также оценка эффективности этих методов при использовании для верификации реальных промышленных программных систем.

Таким образом, можно выделить следующие **цель** и **задачи** данной работы.

Цель и задачи работы

Цель работы — разработка и реализация методов моделирования памяти Си-программ, адаптированных для модулей ядра ОС Linux, для применения в инструментах автоматической статической и дедуктивной верификации, использующих SMT-решатели.

Для достижения цели работы были поставлены следующие **задачи**:

1. Провести анализ существующих методов моделирования памяти Си-программ в инструментах автоматической статической и дедуктивной верификации;

2. Выявить требования к моделям памяти, наиболее подходящим для применения в инструментах автоматической статической и дедуктивной верификации, используемых на практике для модулей ядра ОС Linux;

3. Разработать модели памяти для практически используемых инструментов автоматической статической и дедуктивной верификации, отвечающие выявленным требованиям;

4. Провести теоретическое обоснование корректности и полноты разработанной модели памяти для инструмента дедуктивной верификации;

5. Реализовать предложенные модели памяти в используемых на практике инструментах верификации;

6. Провести практическое сравнение эффективности разработанных моделей памяти с ранее реализованными моделями, в том числе для выявления направлений дальнейшего развития.

Научная новизна работы

Научной новизной обладают следующие результаты работы:

1. Модель памяти на основе теории неинтерпретируемых функций для автоматической статической верификации Си-программ с использованием предикатных абстракций, уточняемых с помощью интерполяции Крейга;

2. Полная модель памяти с поддержкой переинтерпретации типов указателей и автоматизированного разделения на непересекающиеся области (регионы) для дедуктивной верификации Си-программ;

3. Формализация низкоуровневой семантики практически значимого подмножества языка Си, а также соответствующие формальные доказательства корректности и полноты модели памяти с поддержкой вложенных структур и массивов для дедуктивной верификации Си-программ относительно этой семантики;

4. Доказательство корректности и полноты модели памяти для дедуктивной верификации Си-программ с разделением на непересекающиеся регионы;

5. Метод автоматизированного разделения на непересекающиеся регионы для соответствующей модели памяти, полный (полностью автоматический) для ограниченного класса Си-программ;

6. Доказательство полноты метода автоматического разделения на непересекающиеся регионы при использовании соответствующей модели памяти для ограниченного класса Си-программ.

Теоретическая и практическая значимость

Предложено две модели памяти для использования в инструментах автоматической статической и дедуктивной верификации Си-программ.

Реализация модели на основе теории неинтерпретируемых функций в инструменте статической верификации CPAchecker позволила уменьшить число ложных сообщений об ошибках при верификации модулей ядра ОС Linux, а также расширить соответствующий набор требований корректности, существенно использующих адресную арифметику.

Полная модель памяти с поддержкой вложенных структур и массивов, а также объединений и переинтерпретации типов указателей с автоматизированным разделением на непересекающиеся регионы, была разработана и реализована в инструменте дедуктивной верификации Си-программ Jessie.

Разработанная модель памяти позволила адаптировать инструмент Jessie для дедуктивной верификации модулей ядра ОС Linux. За счет этого в инструменте Jessie удалось обеспечить поддержку вложенных структур и переинтерпретации типов указателей на целочисленные значения. Предложенная модель памяти также расширяет область применимости реализованной ранее в инструменте Jessie модели памяти с регионами, а также позволяет моделировать семантику Си-программ более полно в сравнении с исходной версией Jessie.

Реализованные механизмы моделирования памяти могут использоваться как в исследовательских так и в производственных целях. Они опубликованы под открытой лицензией. Заинтересованными отечественными пользователями этих результатов могут быть такие исследовательские центры как МГУ, СПбГУ, МВТУ, СПбПУ, ФГУ ФНЦ НИИСИ РАН, ИПМ им. М.В. Келдыша РАН, АО “НПО РусБИТех” и др.

Модифицированный инструмент дедуктивной верификации Jessie может быть использован при обучении студентов в рамках курса формальной верификации программ в таких университетах как МГУ, СПбГУ, НИУ ВШЭ и СПбПУ.

Публикации и зарегистрированные программы

Всего по теме диссертации автором опубликовано 10 работ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Основные результаты опубликованы в работах [3, 5, 11, 7]. Все работы опубликованы в изданиях перечня ВАК, 9 работ ([2, 3, 4, 5, 6, 7, 8, 9, 10]) входят в Scopus, работа [7] опубликована в сборнике трудов международной конференции. В работах [3, 5] Хорошилов А.В. консультировал автора работы по вопросам возможностей языка Си, используемым в коде модулей ядра ОС Linux, а также по набору видов свойств, проверяемых для модуля безопасности в рамках проекта Astraver. Работа [11] написана совместно с Мутилиным В.С. на основе выполненной автором данной работы реализации и краткого

формального описания соответствующего метода. В работе [7] автором написан раздел, описывающий предикатный анализ, а также соответствующая реализация построителя формул для этого анализа. В ходе выполнения работы было получено 6 свидетельств о государственной регистрации программ для ЭВМ [1–6, см. стр. 191], среди которых непосредственно автором работы была выполнена реализация четырех программ ([1–3, 6]).

Апробация работы

Основные положения работы докладывались на следующих конференциях и семинарах:

1. Двадцатая международная научно-техническая конференция “International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)” (Гренобль, Франция, 5-13 апреля 2014).

2. Международная научно-практическая конференция “Tools & Methods of Program Analysis” (Кострома, Россия, 14-15 ноября 2014 года).

3. Научно-исследовательский семинар Института системного программирования РАН.

4. Международный симпозиум “International Symposium On Leveraging Applications of Formal Methods, Verification and Validation” (Amirandes, Гераклион, Крит, Греция, 18-20 октября 2010).

5. Научно-исследовательский семинар лаборатории “Software and Computational Systems Lab” университета Пассау, Германия.

Личный вклад

Все представленные результаты были получены автором лично.

Структура и объем диссертации

Работа состоит из введения, четырех глав, заключения, списка литературы (174 наименования) и одного приложения. Основной текст диссертации (без приложений и списка литературы) занимает 165 страниц.

Глава 1

Обзор работ в области моделирования памяти Си-программ в инструментах статической верификации

Прежде чем приступить к рассмотрению моделирования памяти Си-программ в инструментах статической верификации, рассмотрим в общих чертах основные подходы к статической верификации программ и роль инструментов автоматического разрешения проблемы выполнимости логических формул (*решателей*) в этих подходах.

1.1. Обзор методов и инструментов статической верификации

Большинство подходов к статической верификации программ можно условно разделить на три большие группы, различающиеся использованием различных подходов к формализации понятия программы.

1.1.1. Подходы, рассматривающие программы как совокупности потенциальных трасс выполнения

К первой группе можно отнести подходы, рассматривающие программы как совокупности потенциально возможных трасс их выполнения (которые в рамках данных подходов часто называют просто *выполнениями* программы, англ. *execution*), представляющих собой последовательности (возможно, бесконечные) состояний программы, соединенных направленными переходами, соответствующими возможным элементарным изменениям состояния про-

граммы при ее выполнении. Для выполнений программ формулируются два основных вида проверяемых свойств — *безопасности* (англ. *safety*) и *живучести* (англ. *liveness*). Свойства безопасности формулируются в виде множеств состояний программы, которые считаются ошибочными. Таким образом, программа является корректной относительно некоторого свойства безопасности тогда и только тогда, когда среди всех ее возможных выполнений нет выполнений, содержащих хотя бы одно ошибочное состояние. Свойства живучести формулируются только для бесконечных выполнений в виде некоторого условия живучести, которому должно удовлетворять любое бесконечное выполнение. Для условий живучести должно выполняться следующее ограничение: для любого конечного выполнения программы должно существовать некоторое содержащее его в виде префикса бесконечное выполнение программы, удовлетворяющее условию живучести. Таким образом, свойства живучести не могут быть нарушены ни на каком конечном выполнении. Это ограничение позволяет строго отличать свойства безопасности и живучести друг от друга, так как любое свойство безопасности может быть нарушено на некотором конечном выполнении, а любое свойство живучести — нет [12]. Программа является корректной относительно некоторого свойства живучести тогда и только тогда, когда либо все ее выполнения являются конечными, либо любое ее бесконечное выполнение удовлетворяет заданному условию живучести.

Проверка моделей программ

Большинство методов статической верификации программ как наборов возможных трасс выполнения являются разновидностями общего подхода под названием *проверка моделей программ* (англ. *software model checking*). В рамках этого подхода проверка корректности программы относительно заданного свойства безопасности или живучести осуществляется путем полно-

го (но не обязательно явного) перебора некоторого подмножества объектов, так или иначе соотнесенных с состояниями исходной программы (перебор состояний обычно оказывается достаточным для проверки не только свойств безопасности, но и живучести, см., напр. [13]). При этом конкретные методы различаются как способом перебора, так и способом соотнесения множества состояний исходной программы с подмножеством объектов, на котором осуществляется перебор.

В самом простом случае перебор осуществляется непосредственно на множестве состояний исходной программы. Такой метод подходит только для программ с небольшим конечным числом состояний. Множество состояний для перебора также может ограничиваться принудительно, например максимальным числом переходов из начального состояния. При таком ограничении метод верификации становится в общем случае некорректным, но по прежнему позволяет находить некоторые интересные трассы, нарушающие заданные свойства.

Наиболее распространенными среди разновидностей проверки моделей программ являются методы, в которых используется прием абстракции исходной программы. *Абстракцией* называется программа, во множество всевозможных выполнений которой задано отображение (возможно, частичное) из множества всех выполнений исходной программы. В случае задания (существования) полного отображения абстракция называется *корректной* (англ. *sound*). Из практических соображений также выделают *адекватные* абстракции, в для которых любой набор выполнений, взятых по одному из каждого подмножества среди некоторого выделенного набора непересекающихся подмножеств выполнений исходной программы, отображается в набор попарно не совпадающих выполнений абстракции. Для проверки некоторого свойства программы с использованием корректной абстракции необходимо задать соответствующее свойство на множестве выполнений абстракции этой

программы так, чтобы из корректности абстракции следовала корректность исходной программы. В таком случае адекватность абстракции позволяет, в частности, отличить с помощью проверки моделей программу, некорректную относительно заданного свойства, от корректной относительно этого свойства программы. Адекватность при этом рассматривается относительно двух непересекающихся множеств выполнений: на которых нарушается и на которых не нарушается заданное свойство. Неадекватная относительно этих множеств, хотя и корректная абстракция, может отображать все выполнения исходной программы в выполнения абстракции, нарушающие заданное свойство, не позволяя отличить корректную программу от некорректной.

Методы абстракции можно разделить на те, в которых абстракция строится явно, в виде программы на каком-либо языке, и те, в которых она строится неявно, например, в виде графа достижимости абстрактных состояний (также abstract reachability graph/tree, ARG, ART) [14], автомата над алфавитом всех операторов исходной программы [15] или логической формулы, моделирующей осуществимость некоторого ограниченного подмножества путей выполнения исходной программы [16, 17]. Также по способу построения абстракции выделяются методы ручного, автоматизированного и полностью автоматического построения абстракции.

Большинство методов верификации программ частично или полностью автоматизированы и поддерживаются соответствующими реализациями в виде инструментов верификации.

Примерами инструментов, реализующих проверку заданных пользователем явных моделей программ относительно как свойств безопасности, так и свойств живучести, являются SPIN [18] и NuSMV [19, 20, 21, 22]. SPIN осуществляет явный непосредственный перебор в пространстве состояний модели программы, заданной пользователем на специальном языке Promela. В NuSMV представление пространства состояний оптимизировано с использо-

ванием BDD. Кроме этого NuSMV поддерживает представление пространства состояний и проверяемого свойства в виде логической формулы, модели которой соответствуют выполнениям заданной пользователем модели программы. Таким образом, выполнимость формулы соответствует нарушению проверяемого свойства, а задача верификации программы сводится к задаче проверки выполнимости (так называемой задаче ВЫП или SAT).

Один из достаточно широко известных методов автоматизированного построения неявной модели программы — декартовой предикатной абстракции программы относительно заданного пользователем набора предикатов в виде графа достижимости абстрактных состояний, получаемого с использованием решающих процедур, входящих в состав инструмента интерактивного доказательства теорем PVS [23], описан в статье [24] и был реализован в инструменте Invariant checker [25].

Среди методов автоматического построения абстракции выделяются методы, строящие абстракции в виде пропозициональных логических формул или *логических формул в теориях* (SMT-формул, от англ. Satisfiability Modulo Theories), моделирующих осуществимость некоторого ограниченного по глубине подмножества путей выполнения исходной программы, а также методы итеративного построения абстракции для всей программы с помощью уточнения по неосуществимым в исходной программе (фиктивным) контрпримерам (CEGAR, от англ. Counter-Example Guided Abstraction Refinement). К первой группе методов относятся методы ограничиваемой верификации (BMC, англ. Bounded model checking) [16] и k-индукции [26, 27], реализованные, например, в инструментах CBMC [16, 28], ESBMC [29] и LLBMC [30]. Методы итеративного построения абстракции по контрпримерам различаются по способу представления абстракции программы и по способу уточнения абстракции на каждой итерации. Для представления абстракций могут использоваться, например, булевы программы, то есть программы, в которых

все переменные могут принимать лишь два значения [31], графы достижимости абстрактных состояний [14], автоматы над алфавитом всех операторов исходной программы [15]. Для уточнения абстракции по контрпримеру могут использоваться различные эвристические техники (например, [32, 33]), интерполяция Крейга [34], поиск индуктивных инвариантов [35, 36] или даже методы искусственного интеллекта, такие как итерация по стратегиям [37]. Примерами инструментов, реализующих представление абстракции в виде булевой программы и ее уточнение с помощью различных эвристических техник являются SLAM (SLAM2) [38] и SATABS [39]. Уточнение графов достижимости абстрактных состояний с помощью интерполяции Крейга для логических формул, представляющих невыполнимые контрпримеры, реализовано в инструментах BLAST [14] и CPAchecker [40]. Поиск индуктивных инвариантов для Си-программ с помощью метода IC3 [35] был реализован, например, в инструменте KRATOS [36]. Все перечисленные инструменты, реализующие методы BMC или CEGAR, так или иначе используют моделирование путей выполнения программ или других подмножеств отношения перехода между состояниями программы в виде SAT и SMT-формул и применяют инструменты автоматического решения проблемы выполнимости этих формул — SAT- и SMT-решатели.

Среди инструментов автоматической верификации программ, опирающихся на использование понятия трассы выполнения, также можно выделить инструменты проверки моделей параллельных программ, использующие редукцию частичных порядков, такие как Java PathFinder [41] и его модификация — Symbolic PathFinder [42], реализующая символьное выполнение с использованием SAT- и SMT-решателей.

Анализ программ

Многие методы автоматической статической верификации программ как наборов возможных трасс выполнения вместо классической проверки моделей программ реализуют разновидности так называемого *анализа программ* (англ. *program analysis*), который отличается от проверки моделей модульностью верификации и применением менее точных и менее ресурсоемких способов абстракции. Модульность верификации означает, что при анализе программ проверяются свойства, формулируемые для отдельных небольших компонентов верифицируемых программ, таких как функции. При этом соответствующие трассы выполнения также рассматриваются для этих компонентов и, как правило, имеют небольшую длину. В качестве абстракций при анализе программ часто используются графы достижимости для абстрактных состояний, являющихся элементами решетки, образуемой отношением включения множеств соответствующих состояний исходной программы на множестве состояний абстракции. В качестве состояний абстракции могут использоваться подмножества явных значений переменных исходной программы, подмножества интервалов этих значений, системы линейных неравенств над переменными исходной программы, конъюнкции необходимых условий достижимости точек исходной программы, представленные в виде формул в теориях и другие домены. Эти абстрактные домены обладают меньшей точностью по сравнению с доменами, используемыми при проверке моделей программ, например, предикатной абстракцией, но алгоритмическая сложность автоматического построения абстракции для них и количество состояний в получаемых абстракциях, как правило, существенно меньше. Это обуславливает применение анализа программ в инструментах верификации, ориентированных на работу с большими объемами исходного кода. Примеры инструментов анализа программ: Klocwork [43], Coverity [44], Svace [45, 46].

В некоторых инструментах анализа программ SAT-решатели применяются для фильтрации ложных предупреждений. Составные абстрактные домены, в которых множества состояний исходной программы приближаются одновременно несколькими состояниями абстракции с различной точностью, часто используются также и в инструментах проверки моделей программ с целью повышения скорости их работы. Примерами таких инструментов являются BLAST [14] и CPAchecker [40].

1.1.2. Дедуктивная верификация программ

Вторая группа подходов к верификации программ — разновидности *дедуктивной верификации*. В этих подходах программы рассматриваются как наборы функций, представляющих собой композиции конструкций некоторого языка программирования, а проверяемые свойства формулируются в виде *контрактных спецификаций* (пред- и постусловий) проверяемых функций, а иногда также и дополнительных спецификаций для некоторых сущностей используемого языка программирования (например, инвариантов типов данных). Дедуктивная верификация осуществляется модульно, отдельно для каждой функции, с использованием контрактных спецификаций, но не кода вызываемых функций. В основе методов дедуктивной верификации лежит логика Хоара [47], методы индуктивных утверждений и оценочных функций Флойда [48], преобразователи предикатов, такие как слабейшее предусловие [49], а также аксиоматическая семантика. С их помощью корректность программы относительно проверяемых свойств сводится к проверке некоторого конечного набора формальных математических утверждений — *условий верификации* (УВ, англ. *verification condition*, VC), представляющих собой формулы в различных формальных логических системах, таких как пропозициональная или сепарационная [50] логика. Для хранения, преобразования

и разрешения получаемого набора условий верификации используются специальные инструменты, такие как решатели формул в теориях, инструменты интерактивного доказательства теорем (например, Coq [51], Isabelle [52], PVS [23, 53]), а также интегрированные инструменты верификации (например, Why3IDE [54], IDE системы верификации Dafny[55]). При этом процесс построения и разрешения условий верификации, как правило, является частично или полностью автоматизированным. Инструменты дедуктивной верификации различаются поддерживаемыми языками программирования и спецификации, а также используемыми логическими системами и поддержкой дополнительных методов верификации (например, инвариантов типов данных, зависимых типов для быстрого установления некоторых свойств во время типизации программы без генерации условий верификации, методологии локально проверяемых инвариантов (LCI) для верификации параллельных программ с разделяемыми данными и др.). Примеры инструментов дедуктивной верификации: Dafny [56] для одноименного языка программирования и спецификации, Why3 [54, 57] для языка программирования и спецификации Why3ML, Caduceus [58], Jessie [59] и WP [60] для языка Си со спецификациями на языке ACSL [61], KeY [62] для языка программирования Java и языка спецификации JML [63], инструмент VCC [64] для языка Си с собственным языком спецификации, F* [65] для одноименного языка программирования и спецификации, VeriFast [66] для программ на языках Java и Си с собственным языком спецификации, GRASShopper [67] для собственного языка программирования и спецификации — и множество других инструментов. Из пересчисленных инструментов все, кроме KeY, VeriFast и GRASShopper непосредственно используют классическую логику с теориями (SMT-формулы) и соответствующие решатели, VeriFast и GRASShopper основаны на сепарационной логике, но при разрешении условий верификации в сепарационной логике также используют решатели

формул в теориях, KeY основан на динамической логике [68, 69] и использует собственный решатель. Из дополнительных методов верификации среди перечисленных инструментов VCC поддерживает методологию локально проверяемых инвариантов (англ. locally checked invariants, LCI) [70] для верификации параллельных программ, работающих с разделяемыми данными, а инструмент F* — зависимые типы.

1.1.3. Подходы, основанные на денотационной семантике

Третья большая группа подходов к верификации программ основана на рассмотрении программ как математических объектов или, иначе говоря, использовании денотационной семантики, описывающей значение составляющих программу компонентов с помощью сопоставления им математических сущностей — денотаций. Верификация программ, таким образом, сводится к проверке соответствующих утверждений об их денотациях, для чего могут быть использованы средства из различных разделов математики и соответствующие инструменты интерактивного и автоматического доказательства теорем. К этой группе относятся реализации языков программирования с зависимыми типами, таких как Gallina (реализован в системе Coq [51]), Agda [71], Idris [72], а также инструменты интерактивного доказательства теорем, поддерживающие написание программ, такие как Isabelle [52] и PVS [23]. В этих языках программирования и инструментах доказательства формулируемых теорем и вспомогательных утверждений (например, необходимых для успешной типизации) осуществляются либо в полуавтоматическом режиме с помощью применения тактик (процедур обратного логического вывода), либо вручную с помощью непосредственного указания терма, представляющего доказательство утверждения. Ведутся исследования в направлении интеграции SMT-решателей в инструменты интерактивного дока-

зательства теорем [73, 74].

1.2. Использование решателей логических формул в теориях в инструментах статической верификации

Решатели логических формул в теориях широко используются во всех рассмотренных группах инструментов статической верификации, кроме реализаций языков с зависимыми типами и инструментов интерактивного доказательства теорем. В инструментах проверки моделей программ (англ. software model checkers) решатели формул в теориях используются для:

- поиска контрпримеров во всем пространстве состояний модели, представленном в виде логической формулы [22];
- проверки осуществимости трасс выполнения, ведущих к ошибочным состояниям [14];
- проверки k-индуктивности заданного свойства программы [26];
- автоматического построения предикатных абстракций [24, 14];
- автоматического уточнения предикатных абстракций по неосуществимым контрпримерам с помощью построения интерполянтов Крейга [75];
- автоматического поиска индуктивных инвариантов с помощью процедуры индуктивного обобщения в рамках метода проверки моделей программ IC3 [35];

а также других, в том числе вспомогательных, целей. В инструментах статического анализа решатели используются для [42, 76]:

- абстрактной интерпретации с использованием различных абстрактных доменов, в том числе в предикатной абстракции;

- символьного выполнения с целью поиска входных данных, при выполнении на которых программа достигает заданного целевого оператора;
- дополнительной проверки осуществимости путей выполнения для уменьшения числа ложных предупреждений.

В инструментах дедуктивной верификации решатели используются для [54, 55]:

- разрешения (проверки выполнимости) условий верификации;
- поиска контрпримеров для реализаций или спецификаций, не соответствующих друг другу;
- проверки используемых аксиоматических формализаций на непротиворечивость;
- выделения существенно используемых утверждений в контексте условия верификации, например, для поиска причин возникающих противоречий или для упрощения полученного условия верификации с целью повторной проверки с помощью другого решателя.

1.2.1. Общая схема использования SMT-решателей

Несмотря на существенные различия между разнообразными задачами, для решения которых применяются решатели логических формул в теориях, можно выделить некоторый общий подход, который описывает все перечисленные случаи их использования в инструментах статической верификации. Этот подход можно условно изобразить в виде схемы, приведенной на рис. 1.1. Для генерации запросов к решателю используется некоторое внутреннее представление программы или ее фрагмента, использующее сущности соответствующего входного или промежуточного языка программирования

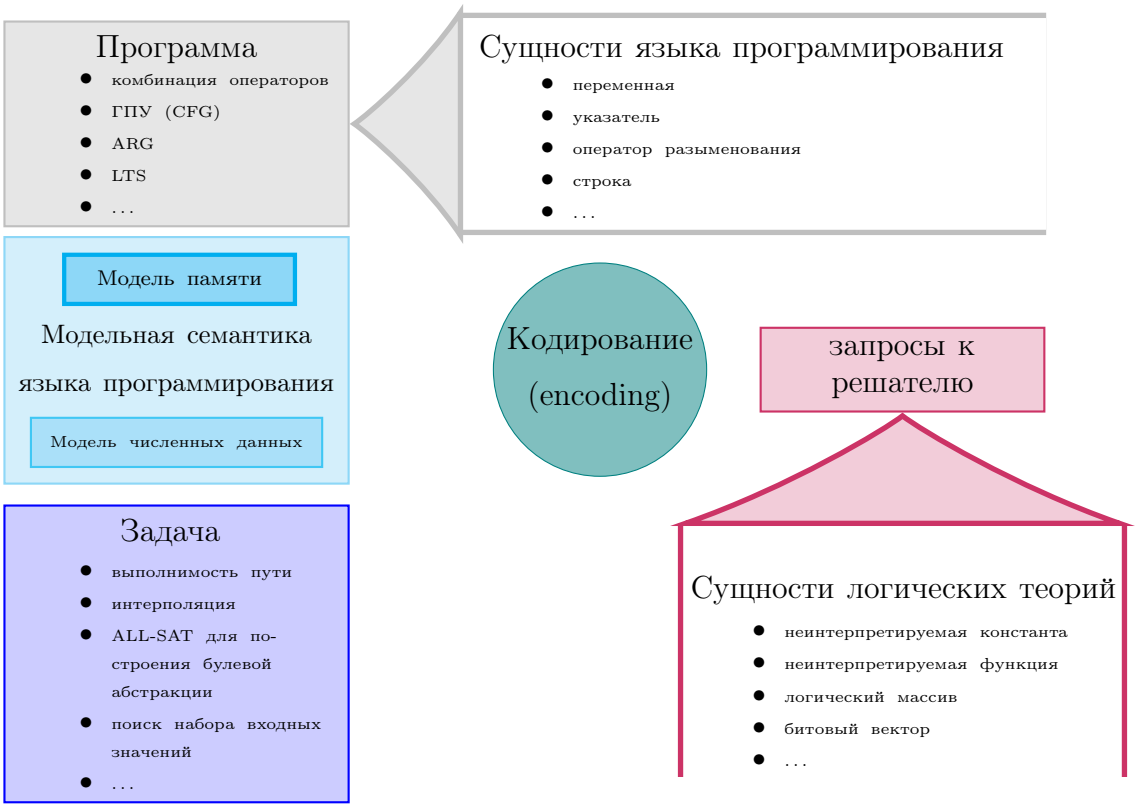

Рис. 1.1. Схема использования решателей логических формул в теориях в инструментах статической верификации

(и, возможно, спецификации), такие как “переменная типа `int`”, “указатель на структуру `struct mutex`”, “оператор разыменования `*`”, “конструктор `None` алгебраического типа `Option`” и т.д. Так как современные решатели применяются не только для проверок выполнимости логических формул, но также, например, для поиска моделей выполнимых формул, поиска интерполянтов Крейга для невыполнимых формул, упрощения формул и других целей, на процесс генерации запросов к решателю влияют типы решаемых задач. Суть же генерации запроса к решателю, как правило, состоит в преобразовании внутреннего представления программы, использующего сущности языка программирования, в некоторое представление логической формулы в теориях,

использующее сущности, определяемые этими теориями, такие как “неинтерпретируемая целочисленная константа”, “неинтерпретируемая функция”, “логический массив”, “битовый вектор” и т.д. Таким образом, имеет смысл ввести условное (неформальное) понятие *модельной семантики языка программирования*, обозначающее способ представления семантики сущностей языка программирования и операций над ними в виде логических формул в теориях. Это понятие с одной стороны отличается от общего понятия *семантики языка программирования*, которое никак не ограничивает используемый способ описания семантики, позволяя выражать её, например, с помощью произвольных определений отношения вычисления [77], произвольных наборов логических утверждений или произвольных денотаций. С другой стороны, оно не ограничивает возможностей использования как произвольного исходного языка программирования, так и произвольного целевого набора логических теорий, что не позволяет определить это понятие достаточно формально. В дальнейшем будем подразумевать, что для используемых в модельной семантике логических теорий существуют автоматические решающие процедуры, возможно, не обладающие полнотой (то есть допускающие незавершение решателя или вердикт “`unknown`” — “`неизвестно`”). Также для уменьшения возможных неоднозначностей будем в дальнейшем называть исходно заданную в произвольном виде семантику рассматриваемого языка программирования его *исходной семантикой*. Процесс преобразования внутреннего представления программы в запрос к решателю, содержащий логическую формулу в теориях, в соответствии с используемой модельной семантикой и типом решаемой задачи будем называть *кодированием* (англ. *encoding*) запроса.

Очевидно, используемая модельная семантика существенно зависит от используемых языков программирования и спецификации. В частности, для императивных языков программирования (то есть основанных на операторах, изменяющих состояние программы), допускающих использование сино-

нимичных (англ. aliasing) указателей или ссылок, например таких как Си, Си++, Java или C#, модельная семантика должна определять способ кодирования операций с указателями или ссылками. Соответствующую часть модельной семантики, содержащую формальные определения понятий указателя или ссылки и формализацию операций над ними с использованием логических формул в теориях, будем называть *моделью памяти*. В данной работе рассматриваются только модели памяти для языка программирования Си в сочетании с различными языками спецификации или другими способами спецификации проверяемых свойств. Рассмотрим аргументы в пользу актуальности задачи статической верификации Си-программ.

1.3. Актуальность задачи статической верификации Си-программ

Актуальность задачи статической верификации Си-программ можно условно рассматривать с трех существенно различных точек зрения:

- актуальность задачи верификации Си-программ как таковых, рассматриваемая безотносительно альтернативных подходов к реализации верифицируемых программ, в частности, задача верификации существующих Си-программ;

- актуальность задачи верификации Си-программ по сравнению с альтернативными реализациями тех же программ на языках более высокого уровня, в частности, на языках, имеющих денотационную семантику (таких как Haskell, Gallina (в системе Coq [51]), Idris [72]) и языках, адаптированных для дедуктивной верификации (таких как Eiffel [78], F* [65], Why3 [54]);

- актуальность задачи верификации Си-программ по сравнению с их альтернативными реализациями на языках того же уровня с сильной статической типизацией, позволяющей проверять корректность создаваемых программ относительно наиболее критических свойств корректности работы с ресурсами (памятью, в том числе разделяемой, файловыми дескрипторами и др.) на этапе компиляции (примеры таких языков — ATS [79] и Rust [80]).

Актуальность верификации существующих Си-программ, а также программ, написанных на Си в силу требований совместимости, обоснована в большом числе работ по статической верификации, например, [59, 81]. Основными аргументами являются широта применения этого языка программирования для реализации низкоуровневых систем или их составных частей, в частности, систем поддержки времени выполнения, таких как планировщики и сборщики мусора; критичность традиционно реализуемого на языке Си программного обеспечения, в том числе ядер операционных систем, в частности, ядра Linux; наличие большого объема существующего широко используемого Си-кода, в том числе в критических по надежности (англ. safety) и безопасности (англ. security) системах, таких как системы управления доступом и операционные системы реального времени; слабость ограничений, накладываемых системой типов и семантикой языка Си, обуславливающая высокую вероятность внесения в код ошибок, связанных с неспецифицированным поведением программ (ошибки управления памятью, ошибки синхронизации).

Актуальность использования языка Си для написания низкоуровневых, а также критических по быстродействию систем по сравнению с языками более высокого уровня, лучше подходящими для статической верификации (в частности, благодаря более строгой, легче формализуемой семантике), в основном обусловлена требованиями производительности, а также возмож-

ностью выражения на языка Си произвольных алгоритмов управления ресурсами, такими как память и процессорное время, что особенно важно при реализации операционных систем и систем поддержки времени выполнения, в том числе и для языков более высокого уровня.

В контексте третьей точки зрения, при сравнении языка Си с языками того же уровня, имеющими более строгую систему типов и семантику, можно указать на существенность различия между некоторым фиксированным набором наперед заданных свойств корректной работы с ресурсами (таких как обращение только к выделенным областям памяти, отсутствие висячих указателей и корректность операций освобождения памяти, отсутствие гонок и взаимных блокировок и др.) и произвольным набором свойств функциональной корректности. Языки, позволяющие относительно эффективно (с малыми накладными расходами на аннотирование кода) обеспечивать корректность относительно фиксированного (хотя часто и наиболее существеного, например, по числу вносимых ошибок) набора свойств, неприменимы, либо неэффективны для верификации свойств, не входящих в этот набор. При верификации же наиболее полного набора свойств функциональной корректности накладные расходы на спецификацию свойств корректной работы с ресурсами, как правило, оказываются несущественными по сравнению с усилиями, необходимыми для полной верификации (см. например, [82]). Таким образом, с точки зрения корректности относительно произвольного набора заданных свойств, в том числе полной функциональной корректности, верификация программ на языках с сильной статической типизацией, дающей некоторый набор дополнительных гарантий, по крайней мере с точки зрения размера необходимых спецификаций на практике не является существенно менее сложной, чем верификация аналогичных программ на слабо типизированных языках. Кроме этого, фиксированный набор правил типизации может существенно затруднять реализацию низкоуровневых частей среды выполнения, таких как

подсистема управления памятью или сборщик мусора, в которых происходит слияние свойств корректности работы с ресурсами и функциональной корректности, осложняющее выражение кода в рамках используемых систем типов. С другой стороны, в силу дополнительных ограничений, обеспечиваемых выразительными системами типов, моделирование памяти в соответствующих языках может быть существенно проще, чем в слабо типизированных (например, в силу ограничений на синонимичность ссылок или указателей). Кроме этого, в силу относительной новизны и относительно малой распространенности сильно типизированных языков с выразительными системами типов в области разработки критических систем, вопрос сравнения эффективности моделирования памяти для таких языков и для слабо типизированных языков практически не исследован. Это можно выделить в качестве одного из направлений дальнейшего сравнительного исследования моделей памяти, в том числе разработанных в ходе выполнения данной работы.

1.4. Теории, поддерживаемые современными решателями

Для более точного (хотя все еще неформального) описания понятия модельной семантики языка программирования необходимо рассмотреть основные логические теории, доступные в большинстве современных решателей. Набор поддерживаемых решателем теорий в конечном счете определяет набор сущностей, с помощью которых может быть формализована модельная семантика используемого языка программирования, а алгоритмическая разрешимость или сложность решения задач выполнимости, соответствующих используемым комбинациям теорий, определяет вероятность успешного разрешения (с вердиктом “выполнимо”/“невыполнимо”) генерируемых логических формул и требуемое для этого количество ресурсов — времени и памяти.

Обзор основных теорий, поддерживаемых современными SMT-решателями дан в приложении A. В обзоре для каждой рассматриваемой теории описана грамматика соответствующих формул, аксиоматизация теории, в случаях, где она не является широко известной, а также указана алгоритмическая разрешимость или сложность соответствующих решающих процедур. Также указана алгоритмическая разрешимость наиболее часто используемых комбинаций логических теорий и приведены некоторые характеристики формул в теориях, использованных на практике при решении индустриальных задач, на основе тестовых наборов из индексов SMT-COMP'16 [83, 84, 85] и SAT-Race'15 [86, 87, 88].

В данной работе будут описаны методы моделирования семантики Си-программ с использованием комбинаций теории неинтерпретируемых функций с теориями целочисленной и вещественно линейной арифметики без кванторов (то есть с использованием логик QF_UFLRA или QF_UFLIA, см. приложение A), а также с помощью комбинаций различных теорий с логикой первого порядка и теорией массивов (например, логики AUFLIRA).

1.5. Проблемы моделирования семантики языка Си

Для статической верификации Си-программ с использованием решателей необходима соответствующая модельная семантика этого языка программирования. Её разработка осложняется противоречивостью требований к модельной семантике, обусловленных исходной семантикой языка Си с одной стороны и возможностями современных решателей с другой. Семантика языка Си обладает следующими особенностями:

- В языке Си допускается практически произвольная синонимичность указателей, включая случаи адресации одним из указателей произвольного (в том числе не соответствующего по типу) участка памяти из

области, занимаемой объектом[1] (например, переменной, массивом или структурой), адресуемым другим указателем. Это сильно осложняет моделирование эффектов операций присваивания в память по указателю, требуя в общем случае использование логического (модельного) представления памяти на уровне байтов или машинных слов.

- Язык Си является слабо типизированным, что позволяет интерпретировать области и подобласти памяти как объекты одновременно нескольких различных типов. Примером такой двойной интерпретации является приведение типа указателя и использование объединений. В силу различий во внутреннем представлении различных типов данных (например, знаковых и беззнаковых целых), формализация такой гибкой интерпретации значений в памяти требует в общем случае еще более точного моделирования памяти реальной вычислительной машины (чем логический массив байт), например, с использованием теории битовых векторов;

- Использование точного моделирования операций на уровне битов также требуется для описания семантики побитовых операций, а также семантики операций с битовыми полями в структурах и объединениях;

- Моделирование операций с указателями в языке Си также осложняется произвольной вложенностью выделяемых в памяти объектов друг в друга (например, вложенные структуры и массивы внутри структур), а также возможностью использования адресной арифметики для доступа как к вложенным, так и к объемлющим объектам (например, получение адреса объемлющей структуры с помощью вычитания смещения поля из адреса вложенной структуры);

[1] Здесь и далее под *объектом* в памяти Си-программы понимается произвольная переменная не составного типа, структура, массив или объединение.

- В языке Си используется ручное управление динамической памятью с помощью явного вызова функций выделения и освобождения; это, а также широкое использование нулевых указателей и возможность адресации памяти, доступной только для чтения, с помощью указателей произвольного типа, требует повсеместной проверки корректности доступов к памяти и предусловий функций, используемых для управления памятью;

- Дополнительную сложность при моделировании операций с указателями представляют собой операции приведения указательных типов к целочисленным и обратно, в частности, указатель полученный в результате такого приведения может потенциально адресовать любую область памяти;

- В языке Си практически отсутствуют специальные средства поддержки параллелизма, что обуславливает высокую вероятность внесения ошибок синхронизации при неверном использовании соответствующих механизмов, предоставляемых используемыми библиотеками, что, соответственно, в общем случае требует применения некоторой методологии верификации параллельных программ для проверки корректности использования примитивов синхронизации;

- Несмотря на большую выразительность языка Си, использование различных его возможностей в реальном промышленном коде является крайне неравномерным. Так одно из исследований [89] показало, что к примеру до 99% операций приведения типа в промышленном коде представляют из себя один их частных случаев получения адреса вложенной или объемлющей структуры (для случая вложения в качестве первого поля), либо приведения указателя к типу `void *` или обратно. Соот-

ветственно, сложность при моделировании памяти Си-программ представляет поиск компромисса между выразительностью соответствующих методов моделирования и их эффективностью для большинства случаев реального использования.

Перечисленные особенности семантики языка Си в общем случае требуют моделирования памяти Си-программ с использованием некоторого сочетания теорий массивов и битовых векторов фиксированной длины. Несмотря на то, что в случае использования пропозициональной логики такая комбинация теорий является алгоритмически разрешимой и, более того, достаточно хорошо поддерживается современными решателями (см. строку QF_ABV в табл. A.1), на практике различия в производительности инструментов верификации при использовании различных подходов к кодированию формул (модельных семантик языка Си) в рамках этой комбинации теорий могут быть весьма существенными. Этому имеется несколько причин. В случае инструментов дедуктивной верификации, использующих логику первого порядка, в силу произвольности задаваемых пользователем спецификаций свойств результирующие условия верификации в общем случае уже не попадают в алгоритмически разрешимый фрагмент комбинации теорий массивов и битовых векторов. Кроме этого, исходя из общих знаний об устройстве разрешающих алгоритмов для теорий битовых векторов и массивов, можно предположить что использование теории битовых векторов будет снижать эффективность инструмента верификации при проверке арифметических свойств (по сравнению с теориями линейной и нелинейной вещественной и целочисленной арифметики), а вклад в производительность теории массивов будет зависеть от числа различных индексов, приходящихся на один логический массив (чем меньше индексов, тем выше производительность). Для подтверждения этих предположений рассмотрим наиболее известные из существующих методов

моделирования семантики Си-программ и наиболее известные результаты сравнения эффективности инструментов верификации, реализующих эти методы.

1.6. Моделирование памяти Си-программ

1.6.1. Классификация моделей памяти

В дальнейшем для каждого рассматриваемого подхода к моделированию семантики Си-программ и в частности, соответствующей модели памяти, будем рассматривать следующие характеристики:

- Теории, используемые при моделировании. Выбор используемых теорий непосредственно влияет на эффективность метода моделирования, а также в некоторых случаях и на область его применимости (например, при необходимости поддержки интерполяции Крейга).

- Поддерживаемые возможности языка Си. Многие распространенные методы моделирования семантики языка Си являются в той или иной степени неполными, поэтому имеет смысл говорить о полноте или, иначе говоря, о выразительности рассматриваемых методов. Выразительность метода особенно важна для инструментов дедуктивной верификации, так как они предъявляют более высокие требования к корректности работы инструмента верификации. В то время как при использовании неподдерживаемой возможности инструмент автоматической статической верификации, к примеру, может продолжить работу с выдачей предупреждения о возможности пропуска ошибки (заменив неподдерживаемый фрагмент кода на некоторое приемлемое приближение), в надежде обнаружить одну или несколько ошибок, не связанных с неподдерживаемой возможностью, инструмент дедуктивной верификации должен

давать как можно больше гарантий именно для случая отсутствия обнаруженных ошибок (так как чаще всего применяется к коду, уже прошедшему множество проверок различными инструментами верификации и тестирования).

- Поддержка неограниченных областей памяти. Так как одной из наиболее сложных задач при моделировании семантики языка Си является моделирование операций с указателями, методы моделирования семантики можно разделять по характеристике соответствующих им моделей памяти, в частности, по поддержке моделирования областей памяти в наперед не ограниченного размера. Как и выразительность метода, поддержка неограниченных областей памяти наиболее важна для инструментов дедуктивной верификации.

- Масштабируемость метода моделирования. Под масштабируемостью будем понимать примерное число операторов языка Си, для которых соответствующая формула пути (для свойства достижимости соответствующего состояния программы), полученная с использованием рассматриваемого метода, может быть разрешена хотя бы одним из современных решателей за время порядка 1 с. Масштабируемость становится важной для применения метода моделирования семантики в инструментах автоматической статической верификации, использующих встраивание функций, и таким образом анализирующих длинные последовательности операторов на путях от точки входа до предполагаемого ошибочного состояния.

Так как в данной работе рассматриваются преимущественно методы моделирования памяти Си-программ, будем классифицировать рассматриваемые методы моделирования семантики по соответствующим моделям памя-

ти и, в частности, по поддержке в них наперед не ограниченных областей памяти.

1.6.2. Модели для ограниченных областей памяти

В методах, поддерживающих моделирование памяти не более, чем наперед заданного размера, возможно использование теорий, для которых существуют более эффективные алгоритмы разрешения формул, в частности, отказ от использования теорий массивов и логики первого порядка в пользу неинтерпретируемых функций или констант и пропозициональной логики. Самым простым с точки зрения используемых теорий является моделирование памяти Си-программ с использованием неинтерпретируемых констант, которое, как правило, опирается на результаты работы некоторого алгоритма анализа синонимичных указателей (*алиасов*).

Использование анализа алиасов

Рассматриваемый метод применялся в инструменте автоматической статической верификации BLAST, использующем предикатную абстракцию с уточнением по невыполнимым контрпримерам. Метод описан в статьях [90, 75], а также в [91] и диссертации [81] (глава 4). Суть данного метода можно описать следующим образом. Пусть имеется некоторый алгоритм межпроцедурного нечувствительный к контексту и потоку анализа алиасов, например [92]. Такой анализ алиасов может определять возможную синонимичность указательных выражений глобально для всей анализируемой программы. Пусть для каждого указательного выражения p из множества всех выражений $\mathcal{E}$ в программе в результате работы анализа алиасов определено множество $\mathcal{A}(p)$ указательных выражений, значение которых может совпадать со значением выражения p (для хранения отображения $p \mapsto \mathcal{A}(p)$

можно использовать, к примеру его представление в виде отношения с помощью BDD [93]). Для представления значений всех выражений в исходной программе в рассматриваемом методе предлагается использовать неинтерпретируемые константы. Будем обозначать конечную последовательность неинтерпретируемых констант, представляющую (изменяющееся в зависимости от состояния) значение синтаксического выражения e (рассматриваемого в контексте некоторой функции) через $v_{\theta(e)}(e)$, где θ — изменяющееся отображение синтаксических выражений в текущие индексы в соответствующих последовательностях. Тогда для сильнейшего постусловия [94] оператора присваивания по указателю на простой тип (не структуру, не объединение и не массив) соответствующее правило вывода в крупношаговой [95] модельной семантике (отношение вычисления обозначено через $\Downarrow$) будет выглядеть так:

assign-deref

$$\frac{\begin{array}{c} e\big|\theta \Downarrow \varphi\big|\theta \\ p \text{ — выражение-указатель на простой тип} \qquad e \text{ — чистое выражение} \end{array}}{\begin{array}{c} *p = e\big|\theta \Downarrow \qquad v_{\theta(*p)+1}(*p) = \varphi \wedge eq(\theta, *p, e) \wedge \\ \bigwedge\limits_{p' \in \mathcal{A}(p)} \mathrm{ite}\left(v_{\theta(p)}(p) = v_{\theta(p')}(p'), \begin{array}{l} v_{\theta(*p')+1}(*p') = \varphi \wedge eq(\theta, *p', e), \\ v_{\theta(*p')+1}(*p') = v_{\theta(*p')}(*p') \wedge eq(\theta, *p', *p') \end{array} \right) \\ \big|\theta \ddagger \{*p \mapsto \theta(*p) + 1\} \ddagger equpd(\theta, *p, e) \ddagger \\ \bigcup\limits_{p' \in \mathcal{A}(p)} \{*p' \mapsto \theta(*p') + 1\} \cup equpd(\theta, *p', *p'), \end{array}}$$

где

$\ddagger$ — операция перезаписи отображения, т.е.

$$\begin{aligned} m_1 \ddagger m_2 &= m_2 \cup \{k \mapsto m_1(k) \mid k \in \operatorname{dom} m_1, k \notin \operatorname{dom} m_2\}, \\ eq(\theta, p_1, p_2) &= \bigwedge\nolimits_{\substack{k=1, \\ \{*^k p_1, *^k p_2\} \subset \mathcal{E}}}^{\infty} v_{\theta(*^k p_1)+1}(*^k p_1) = v_{\theta(*^k p_2)}(*^k p_2), \\ equpd(\theta, p_1, p_2) &= \bigcup\nolimits_{\substack{k=1, \\ \{*^k p_1, *^k p_2\} \subset \mathcal{E}}}^{\infty} \{*^k p_1 \mapsto \theta(*^k p_1) + 1, *^k p_2 \mapsto \theta(*^k p_2) + 1\}, \\ *^k p &= \underbrace{* \ldots *}_{k} p, \quad \mathrm{ite}(c, b_1, b_2) = c \wedge b_1 \vee \neg c \wedge b_2 \end{aligned}$$

Здесь при обновлении значения выражения по указателю происходит одновременное обновление всех его возможных синонимов (разыменований указателей с тем же значением), а также всех присутствующих в программе разыменований обновляемого значения и его синонимов, так как записываемое по указателю значение может также являться указателем. При этом происходит увеличение на единицу индексов во всех соответствующих последовательностях. Чтение значения по указателю p представляется в виде формулы непосредственно как $v_{\theta(*p)}(*p)$. Благодаря использованию дополнительного ограничения $\{*^k p_1, *^k p_2\} \subset \mathcal{E}$ (рассматриваются только выражения, присутствующие в исходном коде программы) данный метод позволяет полностью поддерживать указатели на любые простые типы данных, включая указатели (то есть поддерживаются типы вида $\underbrace{*\ldots*}_{k} t$ для любого k). Используя равенство $*\&a = a$ можно также поддерживать взятие адреса у переменной. С помощью ряда приемов, описанных в [75], данный метод может быть расширен на структуры, в том числе вложенные.

При моделировании семантики операций с указателями с использованием неинтерпретируемых констант и анализа алиасов непосредственно используется только теория равенства. Поэтому данный метод может использовать различные теории, в зависимости от используемой модельной семантики операций над значениями. В инструменте BLAST использовалась теория вещественной линейной арифметики и неинтерпретируемых функций (для приближения нелинейных операций), в принципе возможно также использование теорий нелинейной вещественной арифметики, целочисленной арифметики (линейной и нелинейной), теории битовых векторов конечной длины. Поддерживаемые возможности языка Си в данном методе ограничены простыми и структурными типами данных, не поддерживаются массивы, адресная арифметика, объединения, произвольные приведения типов указателей.

Неограниченные наперед области памяти в общем случае (например, массивы переменной длины) не поддерживаются, однако могут поддерживаться произвольные рекурсивные структуры данных, в том числе неограниченного размера. Для этого вместо множества $\mathcal{E}$ выражений в программе необходимо рассмотреть множество всех выражений, значение которых прямо или косвенно доступно на текущем рассматриваемом пути выполнения (например, для пути, в котором имеются операторы $p = q; \ldots; p\text{->}f = q$; и поле f имеет тот же тип, что и указатель p, доступно не только значение выражения $p\text{->}f$ (прямо), но и $p\text{->}f\text{->}f$ (косвенно)). В силу конечности длины пути, это множество также конечно. В силу использования теорий, эффективно поддерживаемых современными решателями, данный метод хорошо масштабируем, из данных, приведенных в статье [96], можно сделать вывод, что реализация инструмента BLAST, использующая данный метод, успешно работала для трасс длиной в среднем около 1,5 тысяч операторов (включая объявления).

Использование неинтерпретируемых функций

В статье [97] описан метод моделирования семантики предикатов с указателями, используемый в инструменте верификации SLAM2 [38]. В методе предлагается использование теории неинтерпретируемых функций в сочетании с логикой первого порядка. При кодировании используются три неинтерпретируемые функции, A, V и S, соответствующие трем возможным операциям над *размещениями* (англ. *locations*) —произвольными объектами в памяти программы, а именно: взятие адреса размещения ($A(X)$), взятие значения размещения ($V(X)$) и получение вложенного размещения для составных объектов (структур, массивов) ($S(X, f)$, где f — поле или индекс), а также указателей (разыменование указателя выражается как частный случай вложения — $S(X, D)$, $D \neq f$ для любого f). Семантика неинтер-

претируемых функций формализуется с помощью набора аксиом, например $\forall X, Y\,.\; A(X) = A(Y) \implies X = Y$. Метод применим для переменных простых типов, в том числе указателей, структур, в том числе вложенных, и массивов. Не поддерживаются объединения и произвольные приведения типов указателей. В статье [97] не описывается моделирование семантики оператора присваивания (например, в виде соответствующего слабейшего предусловия или сильнейшего постусловия), что не позволяет однозначно сделать вывод о наборе поддерживаемых конструкций языка Си, однако в статьях [38, 98] упоминается использование анализа алиасов, что наиболее вероятно означает, что выразительность данного метода аналогична выразительности метода с использованием неинтерпретируемых констант и анализа алиасов. Таким образом, вероятно, поддерживаются рекурсивные структуры данных неограниченного размера и массивы по крайней мере фиксированной длины. Исходя из данных, приведенных в статье [99], масштабируемость модели памяти, используемой SLAM, сравнима с масштабируемостью модели памяти, используемой BLAST (средняя длина трасс составляет около 120 операторов, не включая объявления).

1.6.3. Модели для неограниченных областей памяти

Модели памяти, поддерживающие произвольные области памяти наперед не ограниченного размера (в частности, массивы переменной длины), опираются на использование теории массивов или логики первого порядка. Рассмотрим последовательно несколько моделей памяти, использующих теорию массивов, но все более минимизирующих число существенно используемых индексов, приходящихся на один массив. Затем рассмотрим модели памяти, использующие неинтерпретируемые функции и логику первого порядка.

Нетипизированная модель памяти

Рассмотрение моделей памяти, использующих теорию массивов, начнем с нетипизированной модели памяти, гипотетической в том смысле, что такая модель памяти не была реализована ни в одном используемом на практике инструменте верификации. В этой модели вся память Си-программы представляется одной последовательностью логических массивов, представляющих текущее состояние памяти программы в виде отображения адресов байт памяти в соответствующие значения. При таком способе моделирования памяти, к примеру, правило вывода для сильнейшее постусловие оператора присваивания по указателю на целое типа `int` будет выглядеть следующим образом:

assign-deref-int

$$\frac{e\big|\theta \Downarrow \varphi\big|\theta \quad p\big|\theta \Downarrow \psi\big|\theta}{p \text{ — выражение-указатель на } \texttt{int} \qquad e \text{ — чистое выражение}}$$

$$*p = e\big|\theta \Downarrow M_{\theta+1} = M_\theta \quad \big[\psi \leftarrow \varphi[0,..,7]\big]\big[\psi+1 \leftarrow \varphi[8,..,15]\big] \big[\psi+2 \leftarrow \varphi[16,..,23]\big]\big[\psi+3 \leftarrow \varphi[24,..,31]\big]\big|\theta+1$$

Здесь предполагается использование порядка байт от младшего к старшему (little-endian), последовательность состояний памяти обозначена через M_θ. Нетрудно видеть, что такая модель памяти обладает очень большой выразительностью по сравнению с ранее рассмотренными моделями, в частности она позволяет поддерживать любые структуры данных, любые массивы и произвольные приведения типов указателей. В принципе, в такой модели памяти при дополнительном использовании теории битовых векторов возможно полное моделирование семантики языка Си (по крайней мере, для фиксированной модели оборудования). Однако отсутствие инструментов верификации, непосредственно реализующих данную модель памяти с большой вероятностью говорит о её плохой масштабируемости. Одной из причин

неэффективности нетипизированной модели памяти является необходимость моделирования большого числа явных ограничений на непересечение адресов различных объектов в памяти программы, что создает трудности при модульной (например, дедуктивной) верификации. В качестве другой причины может быть названо использование большого числа индексов и операций обновления для единственного логического массива, а также необходимость использования теории битовых векторов для моделирования всех операций, в том числе сложения и умножения на число. Далее будут рассмотрены четыре модели памяти, являющиеся в некотором смысле оптимизированными версиями нетипизированной модели памяти.

Типизированная модель памяти

Типизированная модель памяти реализована, в частности, в инструменте дедуктивной верификации VCC2 [64] и описана в статье [100]. Основная особенность этой модели памяти в том, что в предположении об использовании в инструменте дедуктивной верификации возможно делать некоторые дополнительные предположения о семантике моделируемых операторов исходной программы без потери общности. Это возможно благодаря наличию задаваемых пользователем аннотаций, а также возможности генерации нескольких условий верификации вместо одной общей формулы пути. В частности, можно потребовать от пользователя аннотировать некоторые дополнительные аспекты семантики верифицируемой программы (вдобавок к спецификациям, определяемым методологией верификации, таким как инварианты циклов или типов данных), осуществить моделирование семантики в заданных пользователем предположениях, а затем помимо обычных условий верификации сгенерировать дополнительные проверки для заданных пользователем дополнительных предположений. При этом при моделировании памяти в кон-

тексте дедуктивной верификации помимо собственно состояния памяти программы можно использовать также и дополнительное *модельное* (англ. *model* или *ghost*) состояние, изменяемое как в результате выполнения операторов исходной программы, так и в результате выполнения специальных модельных операций, задаваемых в виде аннотаций. В модели памяти VCC2 вводится модельное отображение пар вида (*адрес*, *тип*) в специальный булевый флаг *валидности*, который определяет, является ли данный адрес адресом начала объекта соответствующего типа. Так как Си является слабо типизированным языком, модельное отображение валидности типизированных указателей в VCC2 имеет состояние, которое может быть изменено с помощью специальных модельных операций. Таким образом, модельное отображение валидности остается постоянным при моделировании семантики одного оператора, но может изменяться в ходе выполнения программы. При этом для обеспечения корректности моделирования памяти поддерживается инвариант о том, что в каждом состоянии для каждого адреса может быть не более одного соответствующего валидного типа данных (в силу отсутствия непосредственного моделирования защиты памяти в модельной семантике VCC2 можно считать, что для каждого адреса этот тип всегда ровно один). Моделирование семантики программы происходит в предположении о том, что каждый указатель содержит адрес, валидный тип которого совпадает с типом указателя. Такой указатель будем называть *валидным*. Таким образом, моделируемая программа должна обращаться к памяти только через валидные указатели. Это свойство проверяется с помощью генерации соответствующих условий верификации. В VCC2 используется две модельных операции для изменения состояния отображения валидности, одна для разбиения произвольного объекта в памяти в массив байт (`char`), а другая —для соединения массива байт в объект произвольного типа соответствующего размера. Семантика этих двух операций включает преобразование значений в моделируемой памяти (типи-

зированных логических массивах) в/из типа `char` в соответствии с побайтовой структурой преобразуемого объекта, при этом для кодирования значений различных типов могут в принципе использоваться различные теории. Кроме этого, модель памяти VCC2 предполагает использование логики первого порядка для задания дополнительных аксиом, ограничивающих возможные пересечения адресуемых объектов в памяти программы случаем вложения. Вместе использование типизации памяти и дополнительных аксиом как увеличивают масштабируемость модели памяти, так и уменьшают накладные расходы на спецификацию дополнительных условий непересечения объектов в памяти.

В статье [64] модель памяти описывается с помощью введения базового ("игрушечного", "toy language") языка описания трасс для дедуктивной верификации (содержащих в том числе проверочные условия) и формализации исходной (нетипизированной) и модельной (типизированной) семантик этого базового языка. При этом трассы на базовом языке представляют собой последовательности операторов, а обе семантики определятся как мелкошаговые [101] с помощью правил вывода отношнений вычисления и рекуррентных соотношений функций означивания чистых выражений. Множество значений последовательностей инструкций принимается состоящим из двух элементов — $\top$ и $\bot$, соответствующих результату проверки условий верификации, $\top$ означает выполненность всех условий верификации, $\bot$ — невыполненность хотя бы одного из них. Состояние вычисления определяется как объединение множества $\{\top, \bot\}$ и множества промежуточных состояний, содержащих оставшуюся часть последовательности операторов и некоторый набор отображений, определяющих состояние реальной и модельной памяти программы после выполнения обработанной части трассы. Таким образом, моделирование памяти программы с помощью SMT-формул описывается неявно, однако достаточно компактно и удобно для доказательства свойств рассматриваемой

модельной семантики (в частности, её корректности относительно исходной семантики). Предлагаемая в данной работе модель памяти для инструмента дедуктивной верификации описывается аналогичным образом. Рассмотрим описанный способ формализации модельной семантики подробнее на примере оператора присваивания значения по указателю $*e_1 = e_2$. Соответствующее правило вывода в исходной семантике (обозначения не совпадают с используемыми в [100], но приближены к используемым при описании модели памяти далее в данной работе):

assign-deref

$$\frac{e_1 : t*}{*e_1 = e_2;\ \ oo\big|E, B \blacktriangleright\ oo\big|E, B\big[\ \{[\![e_1]\!]_E \mathbin{\widehat{+}} i \leftarrow [\![e_2]\!]_E[8 \times i, .., 8 \times i + 7] \mid i \in \mathbb{Z}, 0 \le i < |t|\}\ \big]}$$

Здесь $e_1 : t*$ обозначает, что выражение e_1 имеет тип $t*$ согласно правилам типизации базового языка, E — отображение имен переменных в их значения, таким образом, значения глобальных и локальных переменных моделируются отдельно от остальной памяти программы, B — отображение адресов в памяти программы в значения соответствующих байт, $\blacktriangleright$ — отношение вычисления в исходной семантике, $|t|$ — размер типа t в байтах, $M[\{i \leftarrow v \mid P(i)\}] = M \ddagger \{i \mapsto v \mid P(i)\}$, $[\![\cdot]\!]_E$ — функция означивания, задаваемая рекуррентными соотношениями, например $[\![v]\!]_E = E(v)$, $[\![\&e\text{->}f]\!]_E = [\![e]\!]_E \mathbin{\widehat{+}} \mathit{offset}(f)$, где v — переменная, $\mathit{offset}(f)$ — смещение поля структуры f (в предположении о глобальной уникальности полей). Такие правила вывода достаточно легко интерпретировать как соответствующие логические формулы для сильнейших постусловий соответствующих операторов, при этом конкретный способ представления некоторых операций, например, обновления отображения $M[\{i \leftarrow v \mid P(i)\}]$ или сложения битовых векторов фиксированной длины $\widehat{+}$ не уточняется. В частности, если для обновления отображения использовать теорию массивов, а для моделирования операции $\widehat{+}$ — теорию

битовых векторов, то в частном случае $t = \texttt{int}$ получится точно та же семантика (представление сильнейшего постусловия в виде SMT-формулы), что и в рассмотренной ранее нетипизированной модели памяти. Однако для представления обновления отображения можно использовать, к примеру, логику первого порядка, а для приближения семантики операции $\widehat{+}$ — комбинацию теорий линейной арифметики и неинтерпретируемых функций с логикой первого порядка (то есть определить сложение аксиоматически с использованием неинтерпретируемых функций). На практике в инструментах верификации часто применяются подобные оптимизации, что в частности описано и в статье [100]. В модельной семантике (соответствующее отношение вычисления — $\triangleright$) оператору присваивания $*e_1 = e_2$ соответствуют два правила вывода:

$$\text{assign-deref}\quad \frac{e_1 : t* \qquad T\big((\llbracket e_1 \rrbracket_E, t)\big) = \top}{*e_1 = e_2; oo \big| E, T, M \triangleright oo \big| E, T, M\big[(\llbracket e_1 \rrbracket_E, t) \leftarrow \llbracket e_2 \rrbracket_E\big]}$$

и

$$\text{assign-deref}^{\perp}\quad \frac{e_1 : t* \qquad T\big((\llbracket e_1 \rrbracket_E, t)\big) = \perp}{*e_1 = e_2; oo \big| E, T, M \triangleright \perp}.$$

Так как значение $\perp$ соответствует невыполненности одного из условий верификации, такую пару правил вывода следует интерпретировать в контексте дедуктивной верификации как генерацию двух логических формул — сильнейшего постусловия согласно правилу assign-deref, которое будет являться частью формул для последующих условий верификации, определяющей текущее состояние отображений E, T и M, и условия верификации, проверяющего предусловие $\neg\Big(T\big((\llbracket e_1 \rrbracket_E, t)\big) = \perp\Big)$ в соответствии с правилом assign-deref$^{\perp}$. Предусловия таких “парных” правил вывода должны быть согласованы так, что их дизъюнкция должна быть тождественно истинной, а конъюнкция — тождественно ложной, то есть они должны разбивать все множество возмож-

ных состояний вычисления на (два) непересекающихся подмножества. В общем случае правил вывода для одного оператора может быть более двух, при этом среди результатов вычисления также может быть значение $\top$, при котором отрицание соответствующего предусловия интерпретируется как предположение и добавляется в формулу для сильнейшего постусловия. В приведенных правилах память моделируется с помощью двух отображений — описанного ранее отображения валидности T и отображения M пар вида $(\mathit{адрес}, \mathit{тип})$ в значения соответствующих типов. В формуле сильнейшего постусловия для представления значения отображения M могут быть использованы последовательности логических массивов, отображающих адреса в соответствующие значения по одному для каждого типа. Более полный пример формализации модельной семантики описанным способом можно найти в статье [102] и в разделе 4 данной работы.

Итак, модель памяти VCC2 предполагает использование теории массивов и/или логики первого порядка, а также других теорий для моделирования операций над данными, при этом для каждого типа данных может быть использован отдельный набор теорий. При отсутствии обращений к одной и той же области памяти через указатели различных типов в рамках одного оператора (это ограничение можно считать чисто синтаксическим и легко устранимым с помощью использования дополнительных переменных), модель памяти VCC2 поддерживает все конструкции языка Си и применима для областей памяти произвольного наперед не заданного размера. Однако использование в контексте инструмента дедуктивной верификации говорит о применении модели памяти к трассам лишь небольшого размера, как правило, ограниченных рамками одной функции, то есть длиной порядка нескольких десятков операторов.

Модель Бурстала-Борната

Модель Бурсталла-Борната, также известная как модель “поле как массив” (англ. Burstall-Bornat model, component-as-array model) была предложена и использована в работах [103, 104] в контексте частично автоматизированной дедуктивной верификации. Основная идея этой модели памяти заключается в том, чтобы раздельно представлять состояния отдельных компонентов составных объектов, что для языка Си по сути означает разделение представления состояния памяти программы по полям структур. Представление состояния каждого отдельного поля при описании модели может быть не конкретизировано, так же как и при описании типизированной модели памяти (на практике может использоваться теория массивов или аксиоматизация в логике первого порядка). Основную проблему при описании этой модели памяти для языка Си составляют переменные и поля структур, доступные по указателям, то есть потенциально все переменные и поля, для которых в исходном коде Си-программы присутствует операция взятия адреса `&`. Одним из решений этой проблемы является применение к исходному коду программы нормализующих преобразований, устраняющих адресацию переменных и полей структур с помощью переписывания их в указатели и введения специальных фиктивных структур с одним полем соответствующего типа. Такие преобразования описаны, например, в статьях [105, 106, 5, 107]. Другим возможным решением является введение отдельных отображений для представления состояния адресуемых указателями полей и переменных, например, с использованием разделения по типам аналогично типизированной модели памяти. В рамках самой модели Бурсталла-Борната выбор одного из этих решение не существенен, так как в итоге приводит к построению одинаковых логических формул (адресуемые переменные и поля в любом случае разделяются только по типам), однако при расширении данной модели, например, с

помощью дальнейшего разделения представления состояния памяти, перетипирующие преобразования исходного кода оказываются проще по крайней мере при теоретическом описании полученных моделей. Такие преобразования применяются, например, в инструменте дедуктивной верификации Jessie и используются при описании одной из моделей памяти (для инструмента дедуктивной верификации), предлагаемых в данной работе. Решение без дополнительных преобразований используется, например, в инструменте дедуктивной верификации WP [60], реализующем модель Бурсталла-Борната. Один из вариантов этой модели памяти (с использованием логики первого порядка и неинтерпретируемых функций для аксиоматизации операций обновления отображений и операций над указателями) реализован также в инструменте дедуктивной верификации VCC3 [108]. Для достижения полноты поддержки возможностей языка Си в инструментах дедуктивной верификации модель Бурсталла-Борната может быть расширена модельным состоянием и специальными модельными операциями разбиения и соединения аналогично типизированной модели памяти. Модель Бурсталла-Борната может рассматриваться как частный случай модели памяти для дедуктивной верификации, описанной в данной работе, в которой также используются модельное состояние и операции разбиения/соединения.

Модель Бурсталла-Борната в целом обладает примерно теми же характеристиками, что и типизированная модель памяти, однако обеспечивает в общем случае меньшее количество индексов, приходящихся на одно отображение, представляющее часть состояния памяти программы и, как показывают сравнения моделей памяти, упомянутые далее в этом разделе, может быть более эффективна на практике по крайней мере в контексте дедуктивной верификации.

Модель с регионами

В статье [105] предложено расширение модели памяти Бурсталла-Борната, которое используется в инструменте дедуктивной верификации Jessie (а также в его предшественнике, инструменте Caduceus [58]). Основная идея этого вида анализа заключается в том, что память программы в логическом представлении может разделяться не только по компонентам составных объектов (для языка Си — полям структур), как в модели Бурстала-Борната, но и по *регионам* — непересекающимся множествам указательных выражений, таких, что выражения из разных множеств не могут адресовать пересекающиеся области памяти. При этом эти два критерия разделения состояния памяти могут применяться независимо друг от друга. Формально предложенный в статье [105] подход к моделированию памяти Си-программ (но потенциально также применимый и для таких языков, как Java и C#) включает в себя три составляющие. Первая из них — это набор нормализующих преобразований, применяемых к исходной Си-программе, в результате которого получается программа на базовом языке, в котором в качестве типов переменных и полей структур допускаются только целые числа и указатели на структуры (в частности, структуры не могут непосредственно (без использования указателей) содержать в себе другие структуры или массивы, не допускается использование объединений и приведение типов указателей), а вся память программы является динамической. Вторая составляющая — это специальная система типов для базового языка, в которой типы указателей на структуры параметризованы регионами. При этом регионы указательных типов параметров функций сами являются параметрами функций, что порождает систему типов с параметрическим полиморфизмом, аналогичную классической системе Хендли-Милнера [109]. Основной целью введения системы типов для базового языка является существенное упрощение доказательства теоремы об относи-

тельной корректности соответствующей модели памяти, утверждающей, что корректно типизированная программа на базовом языке имеет одинаковую семантику в модели памяти с регионами и в модели Бурсталла-Борната. Наконец, третьей составляющей подхода является алгоритм вывода типов для базового языка, который в статье [105] приводится без формального доказательства корректности, в предположении, что после вывода типов программа на базовом языке будет транслирована на язык WhyML [57] (имеющий систему типов с параметрическим полиморфизмом) так, что все типы, выведенные для базового языка будут непосредственно выражены с помощью типов языка WhyML в результирующей программе. Таким образом будет осуществлена апостериорная проверка корректности вывода типов для программы на базовом языке.

Для моделирования семантики языка Си с использованием модели памяти с регионами, как и для двух ранее рассмотренных моделей, может использоваться теория массивов или комбинация теории неинтерпретируемых функций с логикой первого порядка, а также другие теории для моделирования операций над данными. Возможности языка Си, поддерживаемые моделью памяти с регионами, в том виде, в котором она описана в статье [105], ограничены возможностями используемого в статье базового языка. Таким образом, модель не поддерживает по крайней мере вложенность объектов, объединения и произвольные приведения типов указателей. В остальном, модель с регионами применима для областей памяти в том числе наперед не ограниченного размера. Масштабируемость модели памяти с регионами незначительно отличается от масштабируемости типизированной модели памяти и модели Бурсталла-Борната, однако модель с регионами обеспечивает в общем случае еще меньшее число индексов на отображение, чем в этих моделях, и предположительно может быть эффективнее на практике.

Фрагменты логики разделения

При верификации программ помимо трудностей, связанных с разработкой инструментов верификации, существенную роль играют также трудности, связанные с использованием этих инструментов. В частности, в дедуктивной верификации, в том числе высоко автоматизированной, существенную роль играет размер спецификаций, необходимых для достаточно полной и эффективной (с точки зрения времени работы инструмента верификации) формализации целевых свойств программ. Одной из часто возникающих в этой связи проблем при спецификации свойств программ на языке Си является формализация условий непересечения адресуемых объектов в памяти программы. Типичный пример представляет из себя формализация условия непересечения адресов элементов односвязного списка, требующая использования индуктивного предиката. Размер спецификации условия непересечения адресов возрастает при увеличении числа используемых односвязных списков и кроме этого трудно сочетается с одновременным использованием других структур данных. Этот пример приведен, в частности, в мотивационной части классической статьи [110], описывающей *логику разделения* [110, 50] — логическую систему, в которой на модели формул накладываются так называемые пространственные ограничения, требующие возможности представления одного из компонентов моделей формул — кучи в виде композиции непересекающихся частей. Пространственные ограничения в логике разделения задаются с помощью специальных пространственных логических связок (англ. spatial connectives), таких как разделяющая конъюнкция и разделяющая импликация. Логика разделения позволяет компактно записывать ограничения на значения указателей в рекурсивных структурах данных с использованием индуктивных предикатов, а также компактно формулировать условия непересечения адресов с помощью разделяющей конъюнкции и эффектив-

но делать выводы о возможных эффектах (изменениях значений в памяти программы) вызовов функций. Основным препятствием для непосредственного использования логики разделения в инструментах верификации является алгоритмическая неразрешимость или высокая сложность (как правило, за пределами NP) разрешающих алгоритмов для большинства фрагментов этой логики (см., например, небольшой обзор существующих подходов в статье [111]). Однако для некоторых фрагментов логики разделения все же существуют достаточно эффективные разрешающие алгоритмы, в том числе предполагающие использование SMT-решателей на конечном этапе разрешения. Одним из таких фрагментов, поддержка которого реализована в инструменте дедуктивной верификации, является логика GRASS (англ. Logic of Graph Reachability with Stratified Sets) [111, 112], реализованная в инструменте верификации GRASSHOPPER [67]. Этот инструмент, как и соответствующий фрагмент логики разделения, поддерживает только не вложенные друг в друга рекурсивные структуры данных (такие как одно- и двусвязные списки, бинарные деревья), в том числе неограниченного размера. В получаемых на конечном этапе генерации условиях верификации используется относительно эффективно (соответствующая задача о выполнимости NEXPTIME-полна [113]) разрешимый фрагмент логики первого порядка — эффективно пропозициональные (англ. effectively propositional) формулы в теории равенства и неинтерпретируемых функций. Результаты, приведенные в статье [67] говорят о том, что соответствующий подход к моделированию семантики (в том числе потенциально и для ограниченного подмножества языка Си) достаточно масштабируем по крайней мере для использования в инструменте дедуктивной верификации.

1.6.4. Сравнение моделей памяти на примере

Сравним вначале рассмотренные модели памяти на примере простой спецификированной программы на языке Си. Для включения в сравнение моделей памяти, не поддерживающих неограниченные области памяти, а также инструментов автоматической статической верификации, поддерживающих только задачи достижимости, рассмотрим задачу, не требующую использования при решении неограниченных областей памяти, а также соответствующую спецификацию корректности без использования модельного состояния (для последующего переформулирования в задачу достижимости). Итак, пусть требуется *отсортировать на месте список из 3-х различных элементов, содержащих по 2 значения типа* `int`*, по возрастанию суммы значений элементов*. При решении будем использовать следующую структуру для элементов списка:

```
1 struct list {
2   int v1, v2;
3   struct list *next;
4 };
```

Для спецификации проверяемого свойства воспользуемся вначале языком спецификации ACSL [61, 114], который поддерживается по крайней мере двумя инструментами, реализующими одну из рассмотренных моделей памяти (WP с моделью Бурсталла-Борната и Jessie с моделью на основе регионов). Так как элементы списка различны, достаточно проверить взаимное вложение множеств их значений перед и после сортировки, а также свойство отсортированности в состоянии после сортировки. Для небольшого конечного числа элементов данные свойства могут быть выражены без использования кванторов:

```
5 /*@ axiomatic Permutation_of_3 {
6   @  predicate is_one_of_3{L1,L2}(struct list *e1,
```

```
7    @                                struct list *e2, struct list *e3,
8    @                                struct list *e) =
9    @   \at(e->v1,L2)==\at(e1->v1,L1) && \at(e->v2,L2)==\at(e1->v2,L1) ||
10   @   \at(e->v1,L2)==\at(e2->v1,L1) && \at(e->v2,L2)==\at(e2->v2,L1) ||
11   @   \at(e->v1,L2)==\at(e3->v1,L1) && \at(e->v2,L2)==\at(e3->v2,L1);
12   @  predicate no_new_elements_3{L1,L2}(struct list *l) =
13   @   is_one_of_3{L1,L2}(l,\at(l->next,L1),\at(l->next->next,L1),l) &&
14   @   is_one_of_3{L1,L2}(l,\at(l->next,L1),\at(l->next->next,L1),
15   @                                       \at(l->next,L2)) &&
16   @   is_one_of_3{L1,L2}(l,\at(l->next,L1),\at(l->next->next,L1),
17   @                                       \at(l->next->next,L2));
18   @  predicate is_permutation_of_3{L1,L2}(struct list *l) =
19   @   no_new_elements_3{L1, L2}(l) && no_new_elements_3{L2, L1}(l);
20   @ }
21   @*/
22 /*@ axiomatic Sorted_3 {
23   @  predicate sorted_3(struct list *l) =
24   @   l->v1 + l->v2 <= l->next->v1 + l->next->v2 <=
25   @                    l->next->next->v1 + l->next->next->v2;
26   @ }
27   @*/
```

В языке ACSL присутствует возможность определять предикаты над значениями параметров в разных состояниях программы (и эта возможность поддерживается инструментами верификации Jessie и WP). Состояния программы обозначаются с помощью меток, принимаемых предикатом в виде специальных параметров, например `{L1, L2}`. Для обращения к состоянию программы используется конструкция `\at(`e`,`L`)`, где e — выражение для вычисления в состоянии с меткой L. Для остальных конструкций, используемых в примере семантика языка ACSL аналогична семантике языка Си. Приведем теперь полную ACSL-спецификацию функции, решающей требуемую задачу, и соответствующую реализацию:

```
28 /*@ axiomatic Distinct {
29   @  predicate distinct(struct list *l1, struct list *l2) =
30   @   l1->v1 != l2->v1 || l1->v2 != l2->v2;
31   @ }
32   @*/
33 /*@ requires \valid(l) && \valid(l->next) && \valid(l->next->next);
34   @ requires \separated(l, l->next, l->next->next);
35   @ requires distinct(l, l->next) && distinct(l->next, l->next->next) &&
36   @          distinct(l->next->next, l);
37   @ assigns *l, *(l->next), *(l->next->next);
38   @ ensures sorted_3(l);
39   @ ensures is_permutation_of_3{Pre, Here}(l);
40   @*/
41 void sort(struct list *l)
42 {
43   struct list *l1 = l, *l2 = l1->next, *l3 = l2->next;
44   int v11 = l1->v1, v12 = l1->v2, v21 = l2->v1, v22 = l2->v2,
45       v31 = l3->v1, v32 = l3->v2;
46   long long s1 = (long long)v11 + v12, s2 = (long long)v21 + v22,
47             s3 = (long long)v31 + v32;
48   if (s1 <= s2) {
49     if (s2 <= s3) return;
50     else {
51       if (s1 <= s3) { l2->v1 = v31; l2->v2 = v32; }
52       else { l1->v1 = v31; l1->v2 = v32; l2->v1 = v11; l2->v2 = v12; }
53       l3->v1 = v21; l3->v2 = v22;
54     }
55   } else {
56     if (s1 <= s3) {
57       l1->v1 = v21; l1->v2 = v22; l2->v1 = v11; l2->v2 = v12;
58       /*@ assert l1->v1 + l1->v2 <= l2->v1 + l2->v2; @*/
59     } else {
60       if (s3 <= s2) { l1->v1 = v31; l1->v2 = v32; }
```

```
61        else { l2->v1 = v31; l2->v2 = v32; l1->v1 = v21; l1->v2 = v22; }
62        l3->v1 = v11; l3->v2 = v12;
63      }
64    }
65  }
```

Здесь конструкций `requires` задает предусловие функции, `assigns` — ее эффект, то есть верхнее приближение множества изменяемых значений в памяти, `ensures` — постусловие, `assert` — проверочное утверждение, `\valid`(p) обозначает разрешение на чтение или запись по указателю p, `\separated`$(p_1, \ldots, p_n)$ — условие непересечения в памяти объектов, адресуемых указателями $p_1, \ldots, p_n$.

Рассмотрим, как будет представлено в виде логической формулы сильнейшее постусловие [94] (или слабейшее предусловие) пути, оканчивающегося проверочным условием в строке 58, при использовании рассмотренных ранее моделей памяти:

```
1    l1 = l; l2 = l1->next; l3 = l2->next;
2    v11 = l1->v1; /* ... */ v32 = l3->v2;
3    s1 = (long long)v11 + v12; /* ... */ s3 = (long long)v31 + v32;
4    (!(s1 <= s2))
5    (s1 <= s3)
6    l1->v1 = v21; l1->v2 = v22; l2->v1 = v11; l2->v2 = v12;
7    /*@ assert l1->v1 + l1->v2 <= l2->v1 + l2->v2; @*/
```

Построим сильнейшее постусловие в модели памяти, использующей предварительный анализ алиасов. В пути присутствуют следующие выражения, включающие операции над указателями: `l1->next`, `l2->next`, `l1->v1`, `l1->v2`, `l2->v1`, `l2->v2`, `l3->v1`, `l3->v2`. Пусть предварительный анализ алиасов выдал следующее отношение возможной синонимичности между этими выражениями: {(`l1->next`, `l2->next`), (`l1->v1`, `l2->v1`), (`l2->v1`, `l3->v1`),

$(\texttt{l1->v2}, \texttt{l2->v2}), (\texttt{l2->v2}, \texttt{l3->v2})\}^{\bar{*}}$, где $^{\bar{*}}$ обозначает рефлексивное симметричное транзитивное замыкание. Пусть в начальном состоянии отображение θ содержит нули для всех выражений и переменных, а функция $v_i(x) = x'_i$, где x' обозначает запись выражения x с удаленными символами "-" и ">" (в данном примере такое задание этой функции не приведет к совпадению имен). Тогда сильнейшее постусловие в модели на основе анализа алиасов имеет вид:

$$
\begin{aligned}
&l1_1 = l_0 \wedge l2_1 = l1next_0 \wedge l3_1 = l2next_0 \wedge \\
&v11_1 = l1v1_0 \wedge \ldots \wedge v32_1 = l3v2_0 \wedge \\
&s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
&\neg(s1_1 \leq s1_1) \wedge \\
&s1_1 \leq s3_1 \wedge \\
&l1v1_1 = v21_1 \wedge \\
&\quad \big(l1_1 = l2_1 \wedge l2v1_1 = v21_1 \vee \neg(l1_1 = l2_1) \wedge l2v1_1 = l2v1_0\big) \wedge \\
&\quad \big(l1_1 = l3_1 \wedge l3v1_1 = v21_1 \vee \neg(l1_1 = l3_1) \wedge l3v1_1 = l3v1_0\big) \wedge \\
&l1v2_1 = v22_1 \wedge \\
&\quad \big(l1_1 = l2_1 \wedge l2v2_1 = v22_1 \vee \neg(l1_1 = l2_1) \wedge l2v2_1 = l2v2_0\big) \wedge \\
&\quad \big(l1_1 = l3_1 \wedge l3v2_1 = v22_1 \vee \neg(l1_1 = l3_1) \wedge l3v2_1 = l3v2_0\big) \wedge \\
&l2v1_2 = v11_1 \wedge \\
&\quad \big(l2_1 = l1_1 \wedge l1v1_2 = v11_1 \vee \neg(l2_1 = l1_1) \wedge l1v1_2 = l1v1_1\big) \wedge \\
&\quad \big(l2_1 = l3_1 \wedge l3v1_2 = v11_1 \vee \neg(l2_1 = l3_1) \wedge l3v1_2 = l3v1_1\big) \wedge \\
&\\
&l2v2_2 = v12_1 \wedge \\
&\quad \big(l2_1 = l1_1 \wedge l1v2_2 = v12_1 \vee \neg(l2_1 = l1_1) \wedge l1v2_2 = l1v2_1\big) \wedge \\
&\quad \big(l2_1 = l3_1 \wedge l3v2_2 = v12_1 \vee \neg(l2_1 = l3_1) \wedge l3v2_2 = l3v2_1\big)
\end{aligned}
$$

В этом постусловии делается предположение об отсутствии арифметических переполнений в предложенной программе и для представления арифметических операций используется теория линейной целочисленной арифметики.

Проверочное условие в модели памяти на основе анализа алиасов имеет вид (с учетом индексов, полученных после трансляции всех присваиваний в пути):

$$l1v1_2 + l1v2_2 \leq l2v1_2 + l2v2_2.$$

Предусловие `\separated(l, l->next, l->next->next)` в модели памяти на основе анализа алиасов будет иметь вид $l_0 \neq l1next_0 \wedge l_0 \neq l2next_0 \wedge l1next_0 \neq l2next_0$. Нетрудно убедиться в том, что в предположении `l` $\neq$ `l->next` ($l_0 \neq l1next_0$), которое следует из предусловия функции `sort`, конъюнкция сильнейшего постусловия рассматриваемого пути с отрицанием проверочного условия невыполнима (вообще говоря, она выполнена и в случае $l_0 = l1next_0$), что говорит о выполненности проверочного условия в предложенной функции для всех ее возможных выполнений.

В статьях [38, 98] говорится, что в инструментах, использующих представление предикатов с указателями с помощью неинтерпретируемых функций [97] вместо сильнейшего постусловия используется слабейшее предусловие, которое представляет собой дизъюнкцию слабейших предусловий для различных возможных случаев разбиения указательных выражений в пути на классы эквивалентности. В рассматриваемом существенное влияние на слабейшее предусловие оказывают только выражения `l1` и `l2`, поэтому имеет смысл рассматривать два случая: `l1` $=$ `l2` и `l1` $\neq$ `l2` (и соответствующие структуры в памяти не пересекаются). Таким образом для рассматриваемого пути и проверочного условия в соответствующей модели памяти получится слабейшее предусловие вида:

$$\begin{aligned}
&V\Big(S\big(S(l,2),0\big)\Big) + V\Big(S\big(S(l,2),1\big)\Big) \leq V\big(S(l,0)\big) + V\big(S(l,1)\big) \wedge \ldots \wedge \\
&\quad \neg\Bigg(V\big(S(l,0)\big) + V\big(S(l,1)\big) \leq V\Big(S\big(S(l,2),0\big)\Big) + V\Big(S\big(S(l,2),1\big)\Big)\Bigg) \vee \\
&V\big(S(l,0)\big) + V\big(S(l,1)\big) \leq V\big(S(l,0)\big) + V\big(S(l,1)\big) \wedge \\
&\quad \neg\Big(S(l,0)\big) + V\big(S(l,1)\big) \leq V\big(S(l,0)\big) + V\big(S(l,1)\big)\Big).
\end{aligned}$$

Здесь для полей структуры `list` используются порядковые индексы (0 для `v1`, 1 для `v2` и 2 для `next`). Предусловие `\separated(l, l->next, l->next->next)` в модели памяти на основе неинтерпретируемых функций будет представлено как $A(l) \neq A\big(S(l,2)\big) \wedge A(l) \neq A\Big(S\big(S(l,2),2\big)\Big) \wedge A\big(S(l,2)\big) \neq A\Big(S\big(S(l,2),2\big)\Big)$. Легко видеть, что эта формула общезначима даже без введения дополнительных предположений и аксиом, что вновь соответствует выполненности соответствующего проверочного утверждения для всех возможных выполнений.

Рассмотрим теперь нетепизированную модель памяти. В общем случае при использовании представления памяти программы в виде логического массива байт и теории битовых векторов фиксированной длины сильнейшее постусловий рассматриваемого пути будет выглядеть следующим образом:

$$
\begin{aligned}
&l1_1 = l_0 \wedge l2_1 = M_0[l1_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{11}] \wedge \\
&l3_1 = M_0[l2_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_0[l2_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_0[l2_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_0[l2_1 \mathbin{\widehat{+}} \widehat{11}] \wedge \\
&v11_1 = M_0[l1_1] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{1}] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{2}] \mathbin{\widehat{\circ}} M_0[l1_1 \mathbin{\widehat{+}} \widehat{3}] \wedge \ldots \wedge \\
&v32_1 = M_0[l3_0 \mathbin{\widehat{+}} \widehat{4}] \mathbin{\widehat{\circ}} M_0[l3_0 \mathbin{\widehat{+}} \widehat{5}] \mathbin{\widehat{\circ}} M_0[l3_0 \mathbin{\widehat{+}} \widehat{6}] \mathbin{\widehat{\circ}} M_0[l3_0 \mathbin{\widehat{+}} \widehat{7}] \wedge \\
&s1_1 = v11_1 \mathbin{\widehat{+}} v12_1 \wedge \ldots \wedge s3_1 = v31_1 \mathbin{\widehat{+}} v32_1 \wedge \\
&\neg(s1_1 \mathbin{\widehat{<}}_S s1_1 \vee s1_1 = s1_1) \wedge \\
&(s1_1 \mathbin{\widehat{<}}_S s3_1 \vee s1_1 = s3_1) \wedge
\end{aligned}
$$

$$
\begin{aligned}
M_1 &= M_0 \ \big[l1_1 + 4 \leftarrow v21_1[0,..,7]\big]\big[l1_1 + 5 \leftarrow v21_1[8,..,15]\big] \\
&\quad \big[l1_1 + 6 \leftarrow v21_1[16,..,23]\big]\big[l1_1 + 7 \leftarrow v21_1[24,..,31]\big] \wedge \\
M_2 &= M_1 \ \big[l1_1 + 8 \leftarrow v22_1[0,..,7]\big]\big[l1_1 + 9 \leftarrow v22_1[8,..,15]\big] \\
&\quad \big[l1_1 + 10 \leftarrow v22_1[16,..,23]\big]\big[l1_1 + 11 \leftarrow v22_1[24,..,31]\big] \wedge \\
M_3 &= M_2 \ \big[l2_1 + 4 \leftarrow v11_1[0,..,7]\big]\big[l2_1 + 5 \leftarrow v11_1[8,..,15]\big] \\
&\quad \big[l2_1 + 5 \leftarrow v11_1[16,..,23]\big]\big[l2_1 + 10 \leftarrow v11_1[24,..,31]\big] \wedge \\
M_4 &= M_3 \ \big[l2_1 + 8 \leftarrow v12_1[0,..,7]\big]\big[l2_1 + 9 \leftarrow v12_1[8,..,15]\big] \\
&\quad \big[l2_1 + 10 \leftarrow v12_1[16,..,23]\big]\big[l2_1 + 11 \leftarrow v12_1[24,..,31]\big].
\end{aligned}
$$

При проверочном условии вида:

$$
\begin{aligned}
&M_4[l1_1 \mathbin{\widehat{+}} \widehat{4}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{5}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{6}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{7}] \mathbin{\widehat{+}} \\
&\quad M_4[l1_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{11}] \mathbin{\widehat{<}}_S \\
&M_4[l2_1 \mathbin{\widehat{+}} \widehat{4}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{5}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{6}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{7}] \mathbin{\widehat{+}} \\
&\quad M_4[l2_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{11}] \vee \\
&M_4[l1_1 \mathbin{\widehat{+}} \widehat{4}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{5}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{6}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{7}] \mathbin{\widehat{+}} \\
&\quad M_4[l1_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_4[l1_1 \mathbin{\widehat{+}} \widehat{11}] = \\
&M_4[l2_1 \mathbin{\widehat{+}} \widehat{4}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{5}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{6}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{7}] \mathbin{\widehat{+}} \\
&\quad M_4[l2_1 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_4[l2_1 \mathbin{\widehat{+}} \widehat{11}]
\end{aligned}
$$

и условии `\separated(l, l->next, l->next->next)` вида

$$
\begin{aligned}
&l1' \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l_0 \vee l_0 \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l1' \wedge \\
&l2' \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l_0 \vee l_0 \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l2' \wedge \\
&l1' \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l2'_0 \vee l2'_0 \mathbin{\widehat{+}} \widehat{2} \mathbin{\widehat{<}}_U l1' \wedge \\
&l_0 \mathbin{\widehat{<}}_U 429\widehat{4967}285 \wedge l1' \mathbin{\widehat{<}}_U 429\widehat{4967}285 \wedge l2' \mathbin{\widehat{<}}_U 429\widehat{4967}285, \\
&l1' = M_0[l_0 \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_0[l_0 \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_0[l_0 \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_0[l_0 \mathbin{\widehat{+}} \widehat{11}], \\
&l2' = M_0[l1' \mathbin{\widehat{+}} \widehat{8}] \mathbin{\widehat{\circ}} M_0[l1' \mathbin{\widehat{+}} \widehat{9}] \mathbin{\widehat{\circ}} M_0[l1' \mathbin{\widehat{+}} \widehat{10}] \mathbin{\widehat{\circ}} M_0[l1' \mathbin{\widehat{+}} \widehat{11}].
\end{aligned}
$$

В данном примере память можно также моделировать в упрощающем предположении о равенстве размеров всех простых типов данных размеру типа `int` и отсутствии арифметических переполнений. В этом случае для построения сильнейшего постусловие пути можно использовать теорию линейной

целочисленной арифметики и оно будет выглядеть следующим образом:

$$
\begin{aligned}
&l1_1 = l_0 \wedge l2_1 = M_0[l1_1 + 8] \wedge l3_1 = M_0[l2_1 + 8] \wedge \\
&v11_1 = M_0[l1_1] \wedge \ldots \wedge v32_1 = M_0[l3_0 + 4] \wedge \\
&s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
&\neg(s1_1 \leq s1_1) \wedge \\
&s1_1 \leq s3_1 \wedge \\
&M_1 \ = M_0 \ \ [l1_1 + 4 \leftarrow v21_1] \wedge \\
&M_2 \ = M_1 \ \ [l1_1 + 8 \leftarrow v22_1] \wedge \\
&M_3 \ = M_2 \ \ [l2_1 + 4 \leftarrow v11_1] \wedge \\
&M_4 \ = M_3 \ \ [l2_1 + 8 \leftarrow v12_1].
\end{aligned}
$$

Проверочное условие принимает вид:

$$M_4[l1_1 + 4] + M_4[l1_1 + 8] \leq M_4[l2_1 + 4] + M_4[l2_1 + 8].$$

А предусловие `\separated(l, l->next, l->next->next)` —

$$
\begin{aligned}
&M_0[l_0 + 8] + 2 < l_0 \vee l_0 + 2 < M_0[l_0 + 8] \wedge \\
&M_0\big[M_0[l_0 + 8] + 8\big] + 2 < l_0 \vee l_0 + 2 < M_0\big[M_0[l_0 + 8] + 8\big] \wedge \\
&M_0[l_0 + 8] + 2 < M_0\big[M_0[l_0 + 8] + 8\big] \vee M_0\big[M_0[l_0 + 8] + 8\big] + 2 < M_0[l_0 + 8].
\end{aligned}
$$

В типизированной модели памяти сильнейшее постусловие будет существенно отличаться только в подвыражениях, соответствующих чтению зна-

чений указательного типа:

$$
\begin{aligned}
& l1_1 = l_0 \wedge l2_1 = int^*_0[l1_1 + 8] \wedge l3_1 = int^*_0[l2_1 + 8] \wedge \\
& v11_1 = int^*_0[l1_1] \wedge \ldots \wedge v32_1 = int^*_0[l3_0 + 4] \wedge \\
& s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
& \neg(s1_1 \leq s1_1) \wedge \\
& s1_1 \leq s3_1 \wedge \\
& int_1 \ = int_0 \ [l1_1 + 4 \leftarrow v21_1] \wedge \\
& int_2 \ = int_1 \ [l1_1 + 8 \leftarrow v22_1] \wedge \\
& int_3 \ = int_2 \ [l2_1 + 4 \leftarrow v11_1] \wedge \\
& int_4 \ = int_3 \ [l2_1 + 8 \leftarrow v12_1], \\
& int_4[l1_1 + 4] + int_4[l1_1 + 8] \leq int_4[l2_1 + 4] + int_4[l2_1 + 8].
\end{aligned}
$$

Здесь в качестве имен логических массивов непосредственно использованы обозначения соответствующих типов данных. Проверочное условие аналогично нетипизированной модели.

В модели Бурсталла-Борната в качестве имен массивов в данном примере можно непосредственно использовать имена полей структуры `list`, а в качестве индексов этих массивов — адреса экземпляров этой структуры:

$$
\begin{aligned}
& l1_1 = l_0 \wedge l2_1 = next_0[l1_1] \wedge l3_1 = next_0[l2_1] \wedge \\
& v11_1 = v1_0[l1_1] \wedge \ldots \wedge v32_1 = v2_0[l3_0] \wedge \\
& s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
& \neg(s1_1 \leq s1_1) \wedge \\
& s1_1 \leq s3_1 \wedge \\
& v1_1 \ = v1_0 \ [l1_1 \leftarrow v21_1] \wedge \\
& v2_1 \ = v2_0 \ [l1_1 \leftarrow v22_1] \wedge \\
& v1_2 \ = v1_1 \ [l2_1 \leftarrow v11_1] \wedge \\
& v2_2 \ = v2_1 \ [l2_1 \leftarrow v12_1], \\
& v1_2[l1_1] + v2_2[l1_1] \leq v1_2[l2_1] + v2_2[l2_1].
\end{aligned}
$$

В предусловии достаточно потребовать три попарных неравенства: $l_0 \neq next_0[l_0] \wedge l_0 \neq next_0\big[next_0[l_0]\big] \wedge next_0\big[next_0[l_0]\big] \neq l_0$.

В модели с регионами будет использоваться три раздельных региона для каждого из трех элементов списка. Имена логических массивов составим из имен регионов (`l1`, `l2` и `l3`) и имен полей структуры `list`:

$$
\begin{aligned}
&l1_1 = l_0 \wedge l2_1 = l1next_0[l1_1] \wedge l3_1 = l2next_0[l2_1] \wedge \\
&v11_1 = l1v1_0[l1_1] \wedge \ldots \wedge v32_1 = l3v2_0[l3_0] \wedge \\
&s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
&\neg(s1_1 \leq s1_1) \wedge \\
&s1_1 \leq s3_1 \wedge \\
&l1v1_1 \quad = l1v1_0 \quad [l1_1 \leftarrow v21_1] \wedge \\
&l1v2_1 \quad = l1v2_0 \quad [l1_1 \leftarrow v22_1] \wedge \\
&l2v1_1 \quad = l2v1_0 \quad [l2_1 \leftarrow v11_1] \wedge \\
&l2v2_1 \quad = l2v2_0 1 \quad [l2_1 \leftarrow v12_1],
\end{aligned}
$$

$$
l1v1_1[l1_1] + l1v2_1[l1_1] \leq l2v1_1[l2_1] + l2v2_1[l2_1].
$$

Предусловие `\separated(l, l->next, l->next->next)` в этой модели памяти предполагается выполненным по умолчанию при выполнении анализа регионов, поэтому уже содержится в приведенной логической формуле для постусловия.

Для рассмотрения модели памяти, реализованной в инструменте GRASSHOPPER, необходимо вначале представить предусловие `\separated(l, l->next, l->next->next)` с использованием логики разделения. Адресуемую функцией `sort` динамическую память (кучу) можно представить в виде совокупности трех разделенных (непересекающихся) областей, соответствующих трем различным экземплярам структуры `list` — элементам списка. Для разделения кучи на три непересекающиеся области в соответствующей формуле (заданной в логике разделения) можно

дважды использовать разделяющую конъюнкцию ($*$). Для размещения в каждой из областей кучи экземпляра структуры `list` можно использовать тернарное (для фрагмента логики разделения, используемого инструментом GRASSHOPPER, с явной поддержкой структур, в изначальной логике разделения это отношение бинарное, см. [50]) отношение адресации $\cdot\,.\,\cdot \mapsto \cdot$, которое ограничивает соответствующую область кучи одним экземпляром структуры, указатель на которую стоит слева от точки. Справа от точки в этом отношении стоит имя поля структуры указательного типа, а справа от стрелки — указатель на адресуемую полем структуру. Таким образом с помощью фрагмента логики разделения, реализованного в инструменте GRASSHOPPER, предусловие `\separated(l, l->next, l->next->next)` можно записать как:

$$l\,.\,next \mapsto l\,.\,next * l\,.\,next\,.\,next \mapsto l\,.\,next\,.\,next *$$
$$l\,.\,next\,.\,next\,.\,next \mapsto l\,.\,next\,.\,next\,.\,next$$

Или с использованием квантора существования:

$$\exists l2, l3, l4 : list\mathbf{.}\ l\,.\,next \mapsto l2 * l2\,.\,next \mapsto l3 * l3\,.\,next \mapsto l4$$

Язык спецификации, используемый в инструменте GRASSHOPPER позволяет использовать в проверочном условии так называемый “чистый” (англ. pure) фрагмент логики разделения, формулам в котором соответствует пустая куча. В таком случае тривиальное ограничение пустоты, накладываемое соответствующей формулой на содержимое кучи, может быть проигнорировано, а соответствующее проверочное условие может быть интерпретировано так же, как в логике без пространственных ограничений. Таким образом, в GRASSHOPPER можно использовать проверочное условие , совпадающее с записанным на языке ACSL. Непосредственно для представления состояния памяти в GRASSHOPPER используется модель памяти Бурсталла-Борната и комбинация теорий неинтерпретируемых функций и линейной целочисленной

арифметики с логикой первого порядка. Однако при использовании логики первого порядка в инструменте GRASSHOPPER делается существенная оптимизация. Для выражения обновления памяти с помощью неинтерпретируемых функций и логики первого порядка часть явно используются аксиомы теории массивов. Однако аксиомы теории массивов не обладают свойством *стратифицированности типов* (англ. *stratified sort restriction*) [115], которое заключается в том, что все новые термы (включая вложенные), получаемые при инстанцировании (подстановке значений переменных под квантором всеобщности) аксиом должны иметь типы, не присутствующие среди типов переменных, связанных квантором всеобщности в какой-либо из аксиом. Если же записать основную аксиому теории массивов с использованием неинтерпретируемых функций:

$$\begin{array}{l} \forall a : \mathit{array}_{i,e} \,.\, i, j : i \,.\, e : e \,. \\ \quad \Big(i = j \implies \mathit{read}\big(\mathit{write}(a, i, e), j\big) = e \Big) \wedge \\ \quad \Big(i \neq j \implies \mathit{read}\big(\mathit{write}(a, i, e), j\big) = \mathit{read}(a, j) \Big) \end{array} ,$$

то легко видеть, что при инстанцировании этой аксиомы возникают новые подтермы ($\mathit{write}(a, i, e)$) типа $\mathit{array}(i, e)$, который присутствует среди типов связанных переменных (a). В таких случаях для соответствующих аксиом в общем случае возможно построить неограниченно большое число экземпляров, применяя только подстановку термов, получаемых при их инстанцировании. Это означает, что основанная на таких подстановках разрешающая процедура для логики первого порядка в таких случаях может не завершаться, и, как следствие, соответствующие формулы могут не попадать во алгоритмически разрешимый фрагмент (для комбинации теорий). Для обеспечения эффективной разрешимости генерируемых формул инструмент GRASSHOPPER использует собственную стратегию явного инстанцирования таких аксиом, обеспечивающую полноту для поддерживаемого им фрагмента логики разде-

ления, и удаляет сами аксиомы из результирующих формул, оставляя лишь достаточный для обеспечения полноты набор их экземпляров [116]. Таким образом получается формула сильнейшего постусловия пути примерно следующего вида (несущественные экземпляры основной аксиомы теории массивов исключены):

$$
\begin{aligned}
&l1_1 = l_0 \wedge l2_1 = read(next_0, l1_1) \wedge l3_1 = read(next_0, l2_1) \wedge \\
&v11_1 = read(v1_0, l1_1) \wedge \ldots \wedge v32_1 = read(v2_0, l3_1) \wedge \\
&s1_1 = v11_1 + v12_1 \wedge \ldots \wedge s3_1 = v31_1 + v32_1 \wedge \\
&\neg(s1 \leq s2) \wedge \\
&(s1 \leq s3) \wedge \\
&v1_1 = write(v1_0, l1_1, v21_1) \wedge \\
&\quad read(v1_1, l1_1) = v21_1 \wedge \\
&\quad \big(l2_1 = l1_1 \vee read(v1_1, l2_1) = read(v1_0, l2_1)\big) \wedge \\
&\quad \big(l3_1 = l1_1 \vee read(v1_1, l3_1) = read(v1_0, l3_1)\big) \wedge \\
&v2_1 = write(v2_0, l1_1, v22_1) \wedge \\
&\quad read(v2_1, l1_1) = v22_1 \wedge \\
&\quad \big(l2_1 = l1_1 \vee read(v2_1, l2_1) = read(v2_0, l2_1)\big) \wedge \\
&\quad \big(l3_1 = l1_1 \vee read(v2_1, l3_1) = read(v2_0, l3_1)\big) \wedge \\
&v1_2 = write(v1_1, l2_1, v11_1) \wedge \\
&\quad read(v1_2, l2_1) = v21_1 \wedge \\
&\quad \big(l1_1 = l2_1 \vee read(v1_2, l1_1) = read(v1_2, l1_1)\big) \wedge \\
&\quad \big(l3_1 = l2_1 \vee read(v1_2, l3_1) = read(v1_2, l3_1)\big) \wedge \\
&v2_2 = write(v2_1, l2_1, v12_1) \wedge \\
&\quad read(v2_2, l2_1) = v12_1 \wedge \\
&\quad \big(l1_1 = l2_1 \vee read(v2_2, l1_1) = read(v2_1, l1_1)\big) \wedge \\
&\quad \big(l3_1 = l2_1 \vee read(v2_2, l3_1) = read(v2_1, l3_1)\big),
\end{aligned}
$$

Проверочное условие принимает вид:

$$read(v1_2, l1_1) + read(v2_2, l1_1) \leq read(v1_2, l2_1) + read(v2_2, l2_1)$$

Однако помимо постусловия пути и проверочного условия инструмент GRASSHOPPER также представляет в виде формулы в используемой комбинации теорий ограничения на содержимое кучи, задаваемые используемыми формулами в логике разделения. Для рассмотренного ранее предусловия будет сгенерирована следующая логическая формула (приведена только существенно используемая подформула):

$$
\begin{aligned}
&\mathit{disjoint}\big(\mathit{singleton}(l1), \mathit{singleton}(l2)\big) \wedge \\
&\mathit{disjoint}\Big(\mathit{union}\big(\mathit{singleton}(l1), \mathit{singleton}(l2)\big), \mathit{singleton}(l3)\Big) \wedge \\
&\big(\forall x : \mathit{list}, X, Y : \mathit{set}(\mathit{list})\,\boldsymbol{.} \\
&\quad \neg \mathit{memeber}(x, X) \vee \neg \mathit{member}(x, Y) \vee \neg \mathit{disjoint}(X, Y)\big) \wedge \\
&\Big(\forall x, y : \mathit{list}\boldsymbol{.} \\
&\quad x = y \wedge \mathit{member}\big(x, \mathit{singleton}(y)\big) \vee x \neq y \wedge \neg \mathit{member}\big(x, \mathit{singleton}(y)\big)\Big) \wedge \\
&\Big(\forall x : \mathit{list}\boldsymbol{.} \\
&\quad \mathit{member}\Big(x, \mathit{union}\big(\mathit{singleton}(l1), \mathit{singleton}(l2)\big)\Big) \wedge \\
&\qquad \Big(\mathit{member}\big(x, \mathit{singleton}(l1)\big) \vee \mathit{member}\big(x, \mathit{singleton}(l2)\big)\Big) \vee \\
&\quad \neg \mathit{member}\big(x, \mathit{singleton}(l1)\big) \wedge \neg \mathit{member}\big(x, \mathit{singleton}(l2)\big)\Big),
\end{aligned}
$$

Здесь для представления ограничений, задаваемых формулой в логике разделения, используется частичная аксиоматизация теории конечных множеств, причем для аксиом, не обладающих свойством стратифицированности типов (в данной формуле — определение операции объединения множеств), также как и для основной аксиомы теории массивов произведено явное инстанцирование (в данном случае — частичное, подставлены только термы типа множества).

Инструмент	Модель памяти	Логика	Alt-Ergo (1.01)		CVC3 (2.4.1)	CVC4 (1.5)	Z3 (4.4.0)
Why3	нетипизированная	QF_ABV	>1800		>1800	>1800	>1800
Why3	нетипизированная	QF_ALIA	165.51	(6344)	0.33	0.45	0.08
Why3	типизированная	QF_ALIA	16.89	(2210)	0.12	0.05	0.02
Why3	Бурсталла-Борната	QF_ALIA	0.19	(121)	0.06	0.02	0.01
Why3	с регионами	QF_ALIA	0.05	(63)	0.01	0.02	0.00
VCC3	∼ Бурсталла-Борната	ALIA				>1800	0.21
Jessie	с регионами	QF_ALIA	0.04	(37)	0.04	0.05	0.01
WP	типизированная	QF_ALIA	69.79	(10555)	0.95	0.14	0.06
GRASShopper	GRASS — фрагмент сепарац. логики	UFLIA				0.07	0.02
CPAchecker	Бурсталла-Борната	QF_UFBV					>1800
CPAchecker	Бурсталла-Борната	QF_UFLIA					0.02
CPAchecker	анализ алиасов в пути	QF_BV					ошибка
CPAchecker	анализ алиасов в пути	QF_LIA					0.05

Таблица 1.1. Время работы решателей, усреднение 5 запусков, с

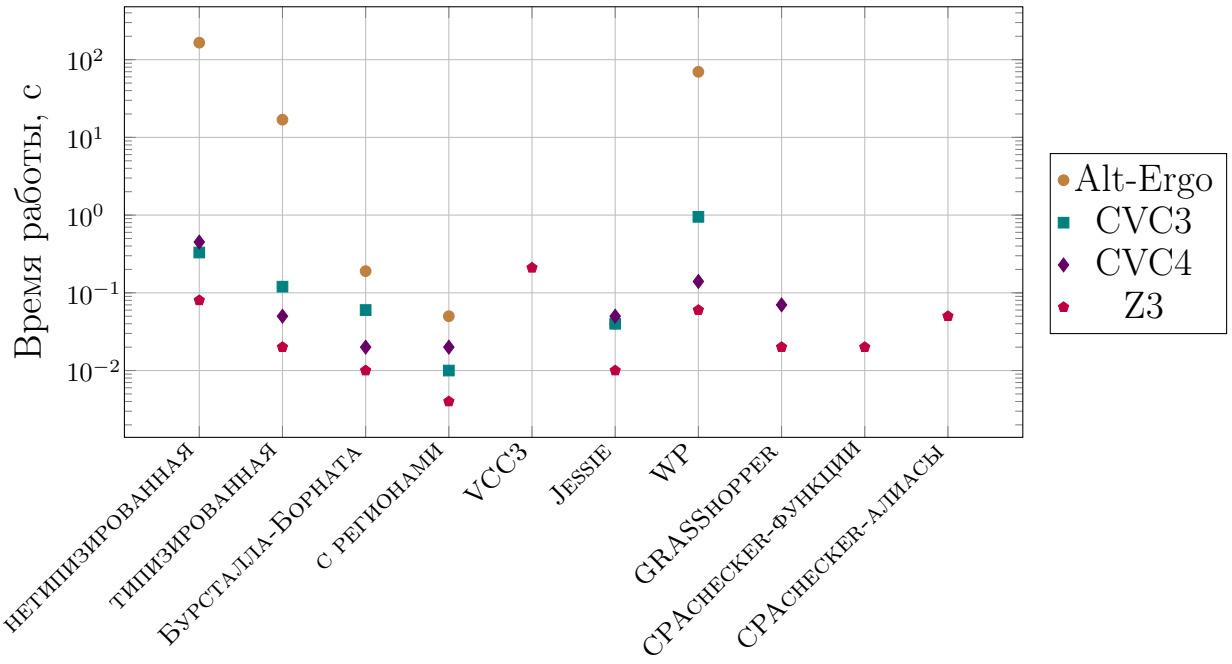

Рис. 1.2. Время работы решателей, усреднение 5 запусков, с

Сравнение времени работы решателей

Сравним теперь время, требуемое на разрешение формул, полученных для рассмотренного примера и моделей памяти с помощью различных инструментов верификации для нескольких широко используемых решателей. Среди инструментов верификации выберем Why3, Jessie, WP, GRASShopper и CPAchecker. Выбранный набор инструментов верификации покрывает все рассмотренные модели памяти, кроме модели на основе неинтерпретируемых функций, используемой инструментом SLAM. В силу особенностей общего алгоритма верификации, используемого инструментом SLAM, при работе на рассмотренном примере он не делает ни одного существенного запроса к SMT-решателю (Z3), однако и не доказывает выполненность постусловия функции `sort` за время 1800 секунд. Будем использовать этот лимит времени работы для всех рассматриваемых инструментов верификации. В инструментах верификации, поддерживающих использование нескольких решателей, при выполнении измерений были использованы решатели Alt-

Ergo [117, 118] версии 1.01, CVC3 [119] (2.4.1), CVC4 [120] (1.5) и Z3 [121] (4.4.0), широко используемые инструментами дедуктивной и автоматической статической верификации. В таблице 1.1 приведены результаты измерения времени работы этих решателей. Моделирование семантики Си и ACSL-спецификаций в инструменте Why3 в различных моделях памяти производилось вручную (на языке Why3ML), также в инструменте GRASShopper вместо Си-программы была использована аналогичная программа на языке программирования и спецификации, который поддерживается этим инструментом. Для решателя Alt-Ergo помимо времени работы также приведено число условных шагов используемого общего алгоритма проверки выполнимости. Для инструментов в таблице указана используемая модель памяти и логика, в которой выражаются результирующие формулы. Логики обозначены с использованием нотации, принятой в соревнованиях SMT-COMP. Использованное процессорное время с точностью до сотых секунды измерялись с использованием средств, предоставляемых самими инструментами верификации (Why3 IDE в случае Jessie и WP). Так как большинство результатов измерения времени довольно велики, различиями, связанными с использованием различных методов измерения при выявлении основных закономерностей можно пренебречь. Отсутствие результата измерения соответствует отсутствию поддержки решателя инструментом верификации, значение >1800 соответствует превышению соответствующего временного ограничения. На рисунке 1.2 приведены времена работы решателей из таблицы 1.1, для времени использована логарифмическая шкала. Как в таблице 1.1, так и на рисунке 1.2 приведены также данные для моделей памяти, разработанных в ходе выполнения данной диссертационной работы. Соответствующие модели памяти описаны в следующих главах.

1.6.5. Известные результаты сравнения моделей памяти

На рассмотренном частном примере хорошо подтвердились предположения о том, что использование теории битовых векторов снижает эффективность инструмента верификации (как оказалось по крайней мере в данном примере, не только при проверке арифметических свойств), а вклад в производительность теории массивов существенно зависит от числа различных индексов, приходящихся на один логический массив (чем меньше индексов, тем выше производительность). Соответствующие выводы, безусловно, в общем случае не применимы к произвольным программам на языке Си и соответствующим проверяемым свойствам. Однако результаты несколько более масштабных экспериментов, приведенные в статье [108] (20 примеров, сравнение числа разрешенных формул в различных моделях памяти и сравнительные графики для конечного списка с возрастающим числом элементов), а именно различие в количестве разрешенных формул до 150 из 1723, в среднем времени разрешения формулы до 3 и более раз и в конкретных временах разрешения формул от 1 до 30 раз, между типизированной моделью с дополнительными условиями разделения адресов и моделью Бурсталла-Борната, а также результаты, приведенные в статье [105], упоминающие увеличение автоматически разрешенных условий верификации с 83.8% до 99.4% при переходе от модели Бурсталла-Борната к модели с регионами, подтверждают сделанное предположение о существенном влиянии числа индексов, приходящихся на логический массив. Предположение о влиянии использования теории битовых векторов подтверждается результатами верификации заданий, полученных на основе драйверов ОС Linux, приведенными далее в данной работе. В отношении результатов рассмотрения моделей памяти на примере, а также приведенных результатов соответствующих исследований можно также отметить, что различия в эффективности (времени разрешения фор-

мул) при использовании различных моделей памяти могут сильно различаться в зависимости от рассматриваемых программ и проверяемых свойств и могут как быть пренебрежимо малыми, так и достигать огромных значений в несколько порядков.

1.7. Выводы

Подведем общие итоги обзора наиболее известных существующих методов моделирования семантики операций над указателями для Си-программ с помощью логических формул в теориях, то есть моделей памяти.

- Модели памяти могут предполагать использование различных комбинаций логических теорий (логик), причем модель памяти может как ограничивать набор возможных комбинаций теорий, так и поддерживать несколько различных комбинаций теорий, возможно с привлечением различных дополнительных предположений.

- В различных моделях памяти делаются различные предположения о верифицируемых программах и проверяемых свойствах. Проверка этих предположений может как предусматриваться в рамках самой модели памяти или соответствующего инструмента верификации, так и нет. Это может влиять на применимость модели памяти в инструменте верификации. Например, инструменты дедуктивной верификации должны делать как можно меньше непроверенных предположений о проверяемой программе.

- Модели памяти различаются по полноте в смысле поддержки конструкций языка Си, а также поддержки неограниченных областей памяти. Соответствующие ограничения также влияют как на применимость модели памяти в инструменте верификации, так и на ее применимость

для решения различных практических задач. Например, инструменты дедуктивной верификации с одной стороны должны как можно более полно поддерживать определенный набор конструкций языка программирования, однако, с другой стороны, при постановке задачи реализации новой верифицированной программы в отличии от задачи верификации существующего кода, набор поддерживаемых конструкций может быть ограничен. В инструментах автоматической статической верификации, верифицирующих практически исключительно существующий код, ограничения на входные программы могут существенно сужать область применимости инструмента, однако на практике часто допускается пропуск ошибок (для небольшого числа проверяемых программ) при использовании в исходном коде программы неподдерживаемых возможностей языка.

- Так как различные наборы логических теорий (логики) обладают разными характеристиками сложности (как правило, от NP-полной до алгоритмически неразрешимой), при использовании логик с большой алгоритмической сложностью, что практически неизбежно в некоторых контекстах, таких как дедуктивная верификация свойств функциональной корректности, большое значение может иметь принадлежность генерируемых формул некоторому фрагменту используемой логики с лучшими характеристиками сложности, а также некоторые другие свойства генерируемых формул, существенно влияющие на производительность разрешающих алгоритмов, такие как количество индексов, приходящихся на логический массив для теории массивов.

В главе 2 данной работы описывается также такой аспект моделей памяти для языка Си как применимость интерполяции Крейга к генерируемым формулам. В таблице 1.2 представлены основные характеристики рассмотренных в

данной главе моделей памяти. Графа “масштабируемость” отражает примерное число существенных строк кода (операторов, инициализирующих объявлений, а также вызовов или возвратов из функций) в одном отрезке простого пути в программе, для моделирования семантики которого соответствующую модель памяти все еще целесообразно использовать на практике из соображений производительности. При превышении указанного числа строк в 2 и более раз следует ожидать существенного падения производительности инструмента верификации (как правило, время верификации начинает возрастать не менее, чем экспоненциально). В таблице приведены в том числе данные для моделей памяти, предложенных в данной работе. Для основных характеристик моделей памяти в таблице приведено одно из трех значений: “да”, “нет” и “частично”. Значение “частично” в графе “рекурсивные структуры” для моделей памяти с анализом алиасов и неинтерпретируемыми функциями соответствует отсутствию возможностей представления в модели памяти соответствующих индуктивных предикатов, а для модели GRASS означает поддержку нескольких видов рекурсивных структур данных без вложенности друг в друга (так называемый “плоский случай”). Значение “частично” в графе “массивы и адресная арифметика” для предложенной модели памяти с неинтерпретируемыми функциями соответствует поддержке массивов не более, чем заданной длины. Значение “частично” в графе “Неограниченные области памяти” для модели памяти на основе анализа алиасов соответствует отсутствию возможности задания предикатов для неограниченных областей памяти. Для графы “Интерполяция Крейга” смысл частичной поддержки объяснен в главе 2.

Модель памяти	Логика	Возможности Си			Неогр. области памяти	Масштаби-руемость, $\leqslant$ LoC	Интерполяция Крейга
		рекурсивные структуры	массивы и адресная арифметика	объединения и приведения типов ук.-лей			
С анализом алиасов	QF_BV, QF_LIA, QF_LRA	част.	нет	нет	част.	3000	да
С неинт. функциями (SLAM)	UFLIA, UFLRA	част.	да	нет	да	3000	нет
С неинт. функциями (предлагаемая)	QF_UFBV, QF_UFLIA, QF_UFLRA	част.	част.	нет	нет	3000	да
(Не)типизированная, Бурсталла-Борната	QF_ABV, QF_ALIA, UFBV, UFLIA	да	да	да	да	25	част.
С регионами	QF_ABV, QF_ALIA, UFBV, UFLIA	да	да	да	да	25	част.
С вложенными регионами	ABV, ALIA, UFBV, UFLIA	да	да	да	да	25	част.
GRASS	UFLIA	част.	нет	нет	да	25	част.

Таблица 1.2. Ограничения моделей памяти

Глава 2

Проблемы существующих моделей памяти для языка Си

Проблемы существующих моделей моделей памяти для языка Си, описанных в главе 1, будем рассматривать в контексте практического применения в двух (одновременно исследовательских и индустриальных) проектах, в рамках которых используются инструменты автоматической статической и дедуктивной верификации.

2.1. Использование инструментов автоматической статической верификации Си-программ в проекте LDV

Целью проекта LDV [122] является разработка набора инструментов для автоматической статической верификации драйверов и других модулей ядра ОС Linux относительно набора разнообразных правил корректности взаимодействия модулей с другими частями ядра (корректности использования предоставляемых внутриядерных интерфейсов) и общих для Си-программ правил корректности (таких как корректность работы с памятью) с применением различных инструментов, а также разработка инструментов для учета найденных ранее ошибок и непрерывного контроля качества модулей ядра (с помощью регрессионной верификации). В рамках проекта LDV осуществляется также поддержка и разработка инструментов статической верификации. Несмотря на то, что в проекте используются различные инструменты статической верификации, в том числе, например, основанные на примене-

нии анализа кучи с помощью образов (англ. shape analysis) [123], на момент написания данной работы для проверки большинства правил корректности использовались инструменты, основанные на использовании предикатных абстракций, такие как BLAST, CPAchecker (в конфигурации, использующей предикатную абстракцию). Успешность применения данного класса инструментов подтверждают как результаты сравнения инструментов статической верификации SV-COMP [124, 125, 126], а именно места от 2 до 3 из 15-20 участвовавших инструментов в соответствующей категории DeviceDrivers64 на каждых соревнованиях с 2012 по 2016 годы включительно, приходящиеся на инструменты данного класса, так и успешное использование инструмента SLAM [127, 38, 99], также основанного на предикатной абстракции с помощью булевых программ, в проекте SDV [128, 129] по статической верификации драйверов ОС Windows.

Инструменты, основанные на использовании предикатных абстракций, как правило (это верно, в частности для BLAST, CPAchecker и SLAM), работают по методу итеративного уточнения абстракции — CEGAR (англ. CounterExample-Guided Abstraction Refinement). В методе CEGAR вводится понятие *точности абстракции*, которое для предикатной абстракции соответствует множеству всех предикатов, используемых в абстракции. Предлагается производить построение абстракции программы итеративно, монотонно увеличивая (в смысле вложения множеств) точность предикатной абстракции на каждой итерации. Одна итерация метода CEGAR состоит из построения корректной абстракции программы с текущей точностью. В результате такого построения возможны три исхода:

- В построенной абстракции нет достижимых ошибочных состояний. Это говорит о том, что текущей точности абстракции оказалось достаточно для доказательства свойства безопасности не только в абстракции, но,

по свойству ее корректности, и в исходной программе. В этом случае основной цикл метода CEAGAR завершается и для программы выдается соответствующий вердикт о выполненности свойства безопасности.

- В построенной абстракции достижимо ошибочное состояние, которое также достижимо и в исходной программе. Это говорит о том, что исходная программа не является корректной относительно заданного свойства безопасности и ее верификацию по методу CEGAR можно завершить с соответствующим вердиктом.

- В построенной абстракции достижимо ошибочное состояние, которое, однако, не достижимо в исходной программе. В этом случае в методе CEGAR предлагается увеличить точность предикатной абстракции, включив в нее новые предикаты так, чтобы в абстракции, построенной с новой точностью, ошибочное состояние не было достижимо по крайней мере по тому же пути, что и в текущей абстракции. Таким образом достигается условный прогресс на каждой итерации метода в том смысле, что из каждой следующей построенной абстракции программы исключается по крайней мере один новый ложный, то есть недостижимый в исходной программе, ошибочный путь.

Полнота метода CEGAR не гарантирована в общем случае, так как количество потенциальных ошибочных путей в исходной программе в общем случае не ограничено, а метод гарантирует исключение только одного пути при каждом уточнении абстракции. Таким образом, на практике количество программ, для которых метод позволяет успешно решать задачу о достижимости будет существенно зависеть от используемого метода уточнения абстракции. Среди методов уточнения абстракции известны синтаксические методы, основанные на выделении новых предикатов из исходного кода программы, например, из условий в операторах условного перехода (`if`) и заголовках цик-

лов; методы, основанные на извлечении невыполнимых подформул (задача UNSAT core) из формул, соответствующих путям выполнения программы; а также методы, основанные на интерполяции Крейга. При этом только методы, основанные на интерполяции Крейга, обладают полнотой в том смысле, что гарантируют недостижимость ошибочного состояния по тому же пути в новой абстракции.

Интерполяция Крейга основана на применении интерполяционной теоремы Крейга-Линдона [130, 131], которая утверждает, что если для двух логических формул φ и ψ общезначимо следствие $\varphi \implies \psi$ и эти формулы имеют хотя бы один общий неинтерпретируемый символ, то существует третья логическая формула ρ, называемая интерполянтом, такая что каждый неинтерпретируемый символ в формуле ρ встречается как в формуле φ, так и в формуле ψ и выполнены следствия $\varphi \implies \rho$ и $\rho \implies \psi$. Эту теорему легко переформулировать для случая невыполнимости конъюнкции $\varphi \wedge \psi$ (подставив вместо ψ выражение $\neg\psi$). В таком случае второе следствие оказывается эквивалентно невыполнимости $\rho \wedge \psi$. Если формула $\varphi \wedge \psi$ соответствует невыполнимой формуле пути к ошибке в исходной программе, такой, что общие неинтерпретируемые символы в формулах φ и ψ соответствуют состоянию программы в некоторой точке этого пути (в которой предполагается вычисление абстракции), то условие $\varphi \implies \rho$ и невыполнимость $\rho \wedge \psi$ гарантируют недостижимость ошибочного состояния в абстракции, включающей предикат ρ. Для практического применения соответствующего метода уточнения абстракции в инструменте верификации достаточно наличие реализации некоторого алгоритма потроения интерполянтов Крейга для невыполнимых конъюнкций формул в некоторой комбинации теорий, такого, что его результирующие интерполянты также являются формулами в этой же комбинации теорий. Такие алгоритмы реализованы в *интерполирующих решателях*, которые могут поддерживать различные комбинации теорий. Однако известно,

что указанным свойством замкнутости относительно интерполяции обладают не все комбинации теорий. В частности, этим свойством не обладает теория целочисленной линейной арифметики и теория массивов. Результирующие интерполянты для этих теорий в общем случае являются формулами в логике первого порядка, в которой многие комбинации теорий, разрешимые в логике без кванторов, могут быть либо не разрешимыми, либо иметь существенно более высокую сложность разрешающих алгоритмов.

Итак, для использования интерполяции Крейга соответствующая модель памяти должна обладать двумя свойствами:

- соответствующие формулы пути должны быть представимы виде конъюнкций вида $\varphi \wedge \psi$, в которых общие неинтерпретируемые символы в φ и ψ соответствуют состоянию программы в точке вычисления абстракции, которая может быт выбрана произвольно;

- модель памяти должна использовать комбинацию теорий, замкнутую относительно интерполяции.

Первое из перечисленных свойств является обязательным условием применимости интерполяции Крейга, в то время как второе влияет на полноту результирующего метода верификации.

В данной работе предлагается использование комбинации теорий вещественной линейной арифметики и неинтерпретируемых функций, которая обладает свойством замкнутости относительно интерполяции. Также вместо вещественной линейной арифметики возможно использование теории целочисленной линейной арифметики, расширенной операциями целочисленного деления и взятия остатка с явно заданным (константным) делителем.

2.2. Проблемы существующих моделей памяти для инструментов автоматической статической верификации Си-программ

Среди существующих моделей памяти, описанных в главе 1 и потенциально применимых для автоматической статической верификации Си-программ с интерполяцией Крейга полностью совместима только модель на основе анализа алиасов (см. табл. 1.2). Модели памяти для неограниченных областей памяти удовлетворяют обязательному условию применимости интерполяции Крейга, но используемые ими комбинации теорий либо не замкнуты относительно интерполяции, либо не полностью поддерживаются интерполирующими решателями (не гарантируется построение интерполянта для любой конъюнкции формул). При этом модель памяти на основе анализа алиасов не поддерживает массивы и адресную арифметику.

Таким образом, цель и соответствующие задачи первой части работы, описанной далее в главе 3, определяются следующим образом. **Цель** — разработка модели памяти, поддерживающей массивы и адресную арифметику для применения в инструменте автоматической статической верификации, использующем интерполяцию Крейга для итеративного уточнения предикатной абстракции по методу CEGAR. Соответствующие **задачи**:

- Разработать модель памяти для предикатного анализа в инструменте CPAchecker, поддерживающую массивы и адресную арифметику и использующую комбинацию теорий линейной арифметики и неинтерпретируемых функций без кванторов.
- Реализовать разработанную модель памяти в инструменте CPAchecker.

- Провести практическое сравнение эффективности разработанной модели памяти с ранее реализованными моделями и, возможно, с альтернативными моделями памяти на основе теории массивов или логики первого порядка, в том числе для выявления направлений дальнейшего развития.

2.3. Использование инструментов дедуктивной верификации в проекте Astraver

Цель проекта Astraver — верификация свойств функциональной корректности модуля безопасности ядра ОС Linux. В рамках проекта осуществляется как верификация формальной математической модели целевого модуля безопасности, так и верификация реализации модуля безопасности (на языка Си) на предмет соответствия ее математической модели. Для верификации Си-кода в проекте используется язык спецификаций ACSL и поддерживающие этот язык спецификаций инструменты дедуктивной верификации WP и Jessie. Выбор языка спецификации ACSL обосновывался его поддержкой в системе Frama-C, которая достаточно хорошо совместима с диалектом GNU C, используемым в ядре Linux, а также свободной доступностью исходного кода этой системы и соответствующих инструментов дедуктивной верификации WP, Jessie и используемого в них обоих инструмента Why3 для модификации и последующей верификации с помощью этих инструментов в том числе закрытого исходного кода.

По сравнению с инструментом WP в инструменте Jessie изначально была реализована модель памяти с регионами, которая как было показано на основе результатов из статьи [105], на практике является более эффективной, так как позволяет уменьшить число индексов, приходящихся на логический массив в результирующих формулах, а также в некоторых достаточно распро-

страненных случаях позволяет избавиться от необходимости явного задания условий непересечения адресов объектов в памяти программы.

Формально описанная в статье [105] модель памяти с регионами включает в себя три составляющие. Первая из них — это набор нормализующих преобразований, применяемых к исходной Си-программе, в результате которого получается программа на базовом языке, в котором в качестве типов переменных и полей структур допускаются только целые числа и указатели на структуры (в частности, структуры не могут непосредственно, без использования указателей, содержать в себе другие структуры или массивы, не допускается использование объединений и приведение типов указателей), а вся память программы является динамической. Вторая составляющая — это специальная система типов для базового языка, в которой типы указателей на структуры параметризованы регионами. При этом регионы указательных типов параметров функций сами являются параметрами функций, что порождает систему типов с параметрическим полиморфизмом, аналогичную классической системе Хендли-Милнера [109]. Основной целью введения системы типов для базового языка является существенное упрощение доказательства теоремы об относительной корректности соответствующей модели памяти, утверждающей, что корректно типизированная программа на базовом языке имеет одинаковую семантику в модели памяти с регионами и в классической модели Бурсталла-Борната. Наконец, третьей составляющей подхода является алгоритм вывода типов для базового языка, который в статье [105] приводится без формального доказательства корректности, в предположении, что после вывода типов программа на базовом языке будет транслирована на язык WhyML (имеющий систему типов с параметрическим полиморфизмом) так, что все типы, выведенные для базового языка будут непосредственно выражены с помощью типов языка WhyML в результирующей программе. Таким образом будет осуществлена апостериорная проверка корректности

вывода типов для программы на базовом языке.

Модель памяти, изначально реализованная в инструменте Jessie [59], является расширением модели памяти с регионами, описанной в статье [105]. По сравнению с исходной моделью в ней добавлена поддержка префиксных приведений типов указателей на структуры и различаемых объединений [106].Восходящее и нисходящее (англ. upcast и downcast), или, иначе, префиксное (англ. prefix) или иерархическое (англ. hierarchical) приведение типа указателя на структуру — это приведение типа указателя на одну структуру к указателю на другую структуру, при котором первое поле одной из структур имеет тип другой структуры (или массива таких структур). Различаемое объединение (англ. discriminated union) — это объединение, поля которого адресуются только через указатель на содержащее их объединение (то есть для которого отсутствуют взятия адресов от полей или соответствующие им приведения типов указателей), и при этом чтение всегда происходит только из тех полей такого объединения, через которые происходила запись в память объединения в самый последний раз.

В ходе применения и адаптации инструмента Jessie для дедуктивной верификации модулей ядра ОС Linux в рамках проекта Astraver [132] было внесено множество более или менее значительных расширений как в саму модель памяти с регионами, описанную в [105, 106] и [5], так и в её реализацию в инструменте Jessie. В контексте верификации кода ядра ОС Linux были выявлены ограничения исходной модели памяти с регионами, и в частности, соответствующего базового языка, основанного на отсутствии вложенных структур, объединений и массивов. В коде ядра ОС Linux вложенные структуры, объединения и массивы и характерные для них конструкции, такие как получение указателя на объемлющую структуру, используются практически повсеместно. В такой ситуации возникла идея изменить базовый язык и его систему типов, включив в них поддержку вложенных структур,

объединений и массивов. В результате оказалось, что все более ранние расширения модели памяти с регионами могут быть выражены в новом базовом языке, расширенном непосредственной поддержкой вложенности, с помощью комбинаций всего двух видов базовых операций: операции получения указателя на вложенную структуру (или массив) и операции получения указателя на объемлющую структуру.

В изначальном расширении модели памяти с регионами префиксные приведения типа и различаемые объединения рассматриваются как отдельные сущности, требующие наличия соответствующих конструкций в базовом языке и соответствующего изменения его системы типов. Поддержка получения указателя на объемлющую структуру (вычитание смещения вложенной структуры относительно начала объемлющей её структуры и последующее приведение типа указателя) и одновременное устранение вложенности при этом требуют применения специальных приемов при построении модели программы на языке Why3ML и большого числа преобразований исходного кода анализируемых программ.

Помимо изначальной неполноты набора поддерживаемых конструкций (относительно набора конструкций языка Си, используемых в промышленном коде), обусловившей необходимость различных расширений базовых языков и моделей памяти, предложенных в статьях [105], [106] и [5], корректность всех этих моделей в данных статьях рассматривается относительно некоторой модельной семантики соответствующего базового языка, для которой не рассматривается проблема её соответствия некоторой низкоуровневой семантике, хотя бы приближенно соответствующей семантике языка Си. Это порождает различные проблемы как с полнотой (например, непрефиксные приведения типов указателей, получение указателя на объемлющую структуру, приведение указателя к соответствующему целочисленному типу), так и с корректностью (например, возможные переполнения значений указателей)

соответствующих моделей памяти при верификации фрагментов промышленного Си-кода.

Для обеспечения более полного покрытия возможностей языка Си и большей совместимости семантики базового языка с соответствующей низкоуровневой семантикой Си, возникла идея формализации двух семантик базового языка — исходной и модельной, аналогично тому, как формализована нетипизированная и типизированная семантики “игрушечного” языка в статье [100], описывающей модель памяти VCC. В отличие от модели памяти VCC, в полученной модели памяти присутствует формализация защиты памяти и разделение модельного состояния памяти на регионы, которое позволяет использовать для эффективного моделирования указателей анализ регионов, аналогичный описанному в статье [105], но адаптированный для базового языка с поддержкой вложенных структур, массивов и объединений.

Таким образом были определены цель и задачи второй части данной работы, описанной в главе 4. **Цель** — разработка модели памяти для инструмента дедуктивной верификации, поддерживающей вложенные структуры и массивы, а также непрефиксные приведения типов указателей и потенциально любые объединения, и обеспечивающей производительность, сравнимую с производительностью ранее реализованной модели памяти с регионами для поддерживаемых в этой модели конструкций. Для достижения цели работы были поставлены следующие **задачи**:

- Разработать модель памяти для инструмента дедуктивной верификации Jessie на основе базового языка с поддержкой вложенных структур и массивов и автоматизированным разделением памяти на непересекающиеся регионы, полностью автоматическим для ранее поддерживаемого класса программ.

- Провести теоретическое обоснование корректности и полноты разрабо-

танной модели памяти относительно формализации низкоуровневой семантики практически значимого (для промышленного кода) фрагмента языка Си.

- Реализовать в инструменте верификации Jessie по крайней мере частичную поддержку вложенных структур и массивов, а также непрефиксных приведений типов указателей.

2.4. Цель и задачи работы

Таким образом, для данной работы в целом можно выделить следующую цель и соответствующие ей задачи. **Цель** — разработка и реализация методов моделирования памяти Си-программ, адаптированных для модулей ядра ОС Linux, для применения в инструментах автоматической статической и дедуктивной верификации, использующих SMT-решатели. Задачи:

- Провести анализ существующих методов моделирования памяти Си-программ в инструментах автоматической статической и дедуктивной верификации.

- Выявить требования к моделям памяти, наиболее подходящим для применения в инструментах автоматической статической и дедуктивной верификации, используемых на практике для модулей ядра ОС Linux.

- Разработать модели памяти для практически используемых инструментов автоматической статической и дедуктивной верификации, отвечающие выявленным требованиям, возможно, на основе существующих методов моделирования памяти.

- Провести теоретическое обоснование корректности и полноты разработанной модели памяти для инструмента дедуктивной верификации.

- Реализовать предложенные модели памяти в используемых на практике инструментах верификации.

- Провести практическое сравнение эффективности разработанных моделей памяти с ранее реализованными моделями, в том числе для выявления направлений дальнейшего развития.

Обзор существующих методов моделирования памяти Си-программ был приведен ранее в главе 1. Требования к моделям памяти для инструментов верификации, используемых на практике для модулей ядра ОС Linux, были определены в данной главе. Разработанные модели памяти, а также результаты применения их реализаций в инструментах верификации рассмотрены в следующих главах.

Глава 3

Модель для ограниченных областей памяти на основе теории неинтерпретируемых функций

Опишем далее предлагаемый в данной работе метод автоматической статической верификации с моделью памяти на основе неинтерпретируемых функций, позволяющий анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы и выражения, содержащие адресную арифметику. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Метод является масштабируемым, так как показывает приемлемые результаты по времени на практически значимом наборе из драйверов устройств ОС Linux.

3.1. Обзор предлагаемого метода

Метод, который предложен в данной работе, основан на использовании теории неинтерпретируемых функций без кванторов. По определению неинтерпретируемая функция — это функция, для которой задано только имя и количество аргументов. Соответственно, для одних и тех же значений аргументов функция выдает один и тот же результат. Неинтерпретируемые функции используются для представления состояния памяти программы как отображения некоторых условных адресов переменных в памяти в их значения. В зависимости от конкретной реализации данный подход позволяет достигать различной точности анализа выражений с указателями, при этом использование теории неинтерпретируемых функций дает возможность его применения только для объектов с наперед заданными размерами. Данный подход облада-

ет существенными ограничениями, так как не применим к последовательностям наперед не заданного числа элементов (например, в списках, деревьях), однако для структур данных относительно небольшого фиксированного размера, часто встречающихся, например, в драйверах устройств ОС Linux, его использование может быть вполне оправдано.

В разработанном методе состояние памяти программы представляется в виде неинтерпретируемой функции f, отображающей некоторые условные адреса переменных в памяти в их значения. Для неинтерпретируемых функций выполнена аксиома конгруэнтного замыкания отношения эквивалентности $\forall a.\ \forall b.\ a = b \implies f(a) = f(b)$. Эта аксиома моделирует равенство значений, полученных после разыменования равных указателей при одном и том же состоянии памяти программы. То есть если есть два равных указателя `p1` и `p2`, то значения `*p1` и `*p2` также равны.

При записи значения по какому-либо адресу (`*e = expr`) происходит смена версии неинтерпретируемой функции, представляющей состояние памяти. При этом в получаемую логическую формулу пути необходимо явно добавлять равенства между значениями соседних версий неинтерпретируемых функций для не участвовавших в присваивании адресов. Поскольку адрес в таком присваивании часто может вычисляться динамически и быть в общем случае неизвестен в точке присваивания, данные равенства в логической формуле будут представлены в виде дизъюнкций следующего вида:

$$e = a \vee f_i(a) = f_{i-1}(a)$$

где e — адресное выражение, a — адрес, для которого выписывается равенство, f_{i-1} и f_i — соответственно старая и новая версия неинтерпретируемой функции, обновляемой в результате присваивания.

Для представления адресов в памяти предлагается использовать суммы вида $b + n$, где b — переменная (неинтерпретируемая константа), соответ-

ствующая адресу некоторой сущности (переменной, структуры, объединения или массива) верхнего уровня, называемая базовым адресом, n — смещение рассматриваемой переменной относительно сущности верхнего уровня. При таком представлении для адресов сущностей верхнего уровня необходимо выполнение условий положительности и непересечения внутренних адресов. Эти условия предлагается записывать в виде двух аксиом модели памяти:

$$b > 0 \qquad (A1)$$
$$B(b + n) = k \quad (A2)$$

где b — переменная, представляющая базовый адрес сущности, k — целое число, уникальное для каждой такой переменной, n — смещение относительно начала сущности, принимающее значения от 0 до $s-1$ включительно, где s — размер сущности.

Экземпляры $(A2)$ позволяют задать всевозможные попарные неравенства внутренних адресов. Докажем, что из $B(b_1 + n_1) = k_1, B(b_2 + n_2) = k_2$ и $k_1 \neq k_2$ следует, что $b_1 + n_1 \neq b_2 + n_2$. Пусть $b_1 + n_1 = b_2 + n_2$, тогда из $(A2)$ получаем $B(b_1 + n_1) = B(b_2 + n_2)$, $k_1 = k_2$ — противоречие с $k_1 \neq k_2$.

Для реализации метода был выбран инструмент верификации CPAchecker, основанный на подходе CEGAR с использованием булевой предикатной абстракции и интерполяции Крейга для получения новых предикатов при уточнении абстракции. Для решения задач проверки выполнимости формулы пути и интерполяции Крейга в CPAchecker используются интерполирующие решатели MathSAT 5, SMTInterpol, Z3, Princess. Все эти решатели полностью поддерживают бескванторные теории вещественной или целочисленной линейной арифметики и равенства с неинтерпретируемыми функциями, но не полностью поддерживают интерполяцию Крейга для теории массивов, в частности и потому, что интерполянты для теории массивов в общем случае могут содержать кванторы всеобщности. Поэтому в рамках

данной работы разрабатывался и исследовался способ построения формулы пути с использованием только неинтерпретируемых функций.

3.1.1. Полнота предлагаемого метода

Одним из важнейших требований, предъявляемых к подходу уточнения и построения абстракции в инструменте, основанном на использовании метода CEGAR, является требование полноты этого подхода, или иначе говоря, требование надежного уточнения абстракции. Оно означает, что на каждой следующей итерации метода CEGAR, после уточнения абстракции на основе какого-либо ложного контрпримера, этот контрпример обязательно оказывается исключенным из уточненной абстракции. Таким образом достигается постоянное уточнение абстракции с точки зрения уменьшения числа порождаемых ею ложных контрпримеров. Для выполнения данного требования в той реализации CEGAR, которая используется инструментом CPAchecker, достаточно выполнения требования индуктивности получаемых интерполянтов и независимого построения различных частей формулы пути.

Независимость построения различных частей формулы пути означает, что каждой из этих частей будет соответствовать одна и та же логическая формула вне зависимости от контекста ее построения (при проверке контрпримера, интерполяции или пересчете абстракции). Это требование могло бы быть нарушено, например, при использовании эвристик, учитывающих наличие либо отсутствие в контрпримере каких-либо неиспользуемых переменных. В предлагаемом подходе такие эвристики не используются. В случае же выполнения обоих требований полноту подхода можно показать, воспользовавшись определением интерполянта. Предлагаемый подход удовлетворяет этим требованиям, так как предполагает последовательное получение индуктивных интерполянтов и для одинаковых блоков операторов строит формулы

пути, совпадающие с точностью до имен индексированных символов.

3.2. Описание метода

3.2.1. Определения

Программы и поток управления

Мы ограничимся рассмотрением простого императивного языка программирования (аналогично [133, 75]), в котором все переменные имеют типы `int` и `int*`, а все операции — это либо присваивания, либо предположения `assume`, представленные в табл. 3.1. Мы рассмотрим программы без вызовов функций, хотя описанный подход может быть расширен на программы с несколькими функциями.

Программа представляется *автоматом потока управления* (АПУ) (от англ. control-flow automaton, CFA). АПУ $A = (L, G)$ состоит из множества точек программы L, моделирующих счетчик команд, и множества дуг потока управления $G \subset L \times Ops \times L$, которые моделируют действия, выполняемые при переходе из одной точки программы в другую. Множество переменных, встречающихся в операциях Ops, обозначим как X. Программа $P = (A, l_0, l_E)$ состоит из АПУ $A = (L, G)$ (моделирующего поток управления программы), начальной точки программы $l_0 \in L$ (моделирует точку входа), и целевой точки программы l_E (моделирует ошибочное состояние).

Конкретное состояние данных программы — это состояние памяти программы, как динамической выделяемой, так памяти под переменные X, выделенные на стеке и в статической памяти. Далее мы не будем разделять разные виды пямяти, а будем считать, что для каждой переменной из множества X выделена память, адрес которой обозначается именем переменной. Состояние памяти задается функцией $f : \mathbb{Z} \to \mathbb{Z}$, отображающей адрес ячейки памяти в

значение, содержащееся в ней. Так как переменные $x \in X$ обозначают адреса ячеек памяти, то значения для переменной x будут представляться как $f(x)$. Обозначим множество конкретных состояний программы как $\mathcal{E}$.

Множества $r \subset \mathcal{E}$ назовем регионами, которые будем представлять с помощью логических формул φ. Формулы будут содержать переменные из множества X, а также неинтерпретируемые функции из множества F, заданные над целыми числами ($\mathbb{Z} \to \mathbb{Z}$), которые служат для моделирования состояния памяти. Формула φ представляет множество $[\![\varphi]\!]$ конкретных состояний данных c, для которых выполнено φ (т. е. $[\![\varphi]\!] = \{c \in \mathcal{E} \mid c \models \varphi\}$).

Конкретное состояние программы — это пара (l, c), где $l \in L$ — точка программы, а c — это конкретное состояние данных. Пара (l, φ) представляет множество $\{(l, c) \mid c \models \varphi\}$ конкретных состояний. Конкретная семантика операции $op \in Ops$ определяется оператором сильнейшего постусловия $SP_{op}(\cdot)$: для формулы φ оператор $SP_{op}(\varphi)$ представляет наименьшее по включению множество состояний, содержащее все состояния, получаемые хотя бы из одного состояния региона, представленного φ, после выполнения оператора op.

Путь σ - это последовательность $\langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ пар из операции и точки программы. Путь σ называется *путем программы*, если он начинается в l_0 (см. определение P) и для каждого i, такого что $0 < i \leq n$, существует дуга АПУ $g = (l_{i-1}, op_i, l_i)$. Таким образом, σ представляет синтаксический путь в АПУ.

Конкретная семантика для пути программы $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ определяется как последовательное применение оператора сильнейшего постусловия для каждой операции: $SP_\sigma(\varphi) = SP_{op_n}(\ldots SP_{op_1}(\varphi)\ldots)$. Формула $SP_\sigma(\varphi)$ называется *формулой пути*.

Множество конкретных состояний, являющихся результатом выполнения пути программы σ представляется парой $(l_n, SP_\sigma(\top))$. Путь программы

называется *достижимым*, если формула $SP_\sigma(\top)$ выполнима. Конкретное состояние программы (l_n, c_n) называется *достижимым*, если существует достижимый путь σ, заканчивающийся в точке l_n, и такой, что $c_n \models SP_\sigma(\top)$. Точка программы l достижима, если существует конкретное состояние c, такое что (l, c) достижимо. Программа корректна (англ. safe), если l_E недостижимо.

Булевы предикатные абстракции

Пусть F — множество неитерпретируемых функций. Пусть $\mathcal{P}$ — множество предикатов из теории без кванторов над переменными программы X и неинтерпретируемыми функциями F. Формула φ — это булева комбинация предикатов из $\mathcal{P}$. *Точность для формул* — это конечное подмножество $\pi \in \mathcal{P}$. *Точность для программы* – это функция $\Pi : L \to 2^{\mathcal{P}}$, которая задает точность для формул в каждой точке программы.

Булева предикатная абстракция $(\varphi)^\pi$ для формулы φ — это сильнейшая булева комбинация предикатов из точности π, которая следует из φ. Данная предикатная абстракция формулы φ, которая представляет регион конкретных состояний программы, используется как *абстрактное состояние данных* (абстрактное представление региона) в верификации программ. Для формулы φ и точности π булева предикатная абстракция $(\varphi)^\pi$ может быть вычислена с помощью запросов к SMT решателю с поддержкой выдачи всех моделей для заданного подмножества булевых переменных (ALL-SAT) следующим образом. Каждому предикату $p_i \in \pi$ сопоставим булеву переменную v_i. Затем сделаем запрос к решателю для выдачи всех векторов решений $v_1, \ldots, v_{|\pi|}$ формулы $\varphi \wedge \bigwedge_{p_i \in \pi} (p_i \iff v_i)$. Для каждого вектора решений мы строим конъюнкцию всех предикатов из π, которые входят в вектор решения как истина. Дизъюнкция всех таких конъюнкций будет булевой предикатной абстракцией для формулы φ.

Абстрактный оператор сильнейшего постусловия с точностью π и операцией op, который преобразует абстрактное состояние φ в следующее состояние φ', может быть определен с помощью применения оператора сильнейшего постусловия и последующего вычисления предикатной абстракции, т. е. $\varphi' = (SP_{op}(\varphi))^{\pi}$. Более детально предикатные абстракции описаны в работах [134, 135, 136].

Кодирование с настраиваемым размером блока

В кодировании с настраиваемым размером блока (Adjustable-Block Encoding, ABE) предикатные абстракции не вычисляются при каждом переходе по дуге из АПУ, а напротив, вычисляются только в некоторых абстрактных состояниях, которые будем называть *состояниями абстракции* (другие абстрактные состояния будем называть состояниями *без абстракции*). На пути между двумя состояниями вычисления абстракции сильнейшее постусловие пути хранится во втором компоненте состояния, который мы назовем *дизъюнктивной формулой пути*. Таким образом, абстрактное состояние ABE содержит два компонента-формулы $\langle \psi, \varphi \rangle$, где формула абстракции ψ — это результат вычисления абстракции, а дизъюнктивная формула φ представляет сильнейшее постусловие с момента вычисления последнего состояния абстракции. Для заданной дуги АПУ $g = (l, op, l')$ и абстрактного состояния $\langle \psi, \varphi \rangle$, следующее состояние либо только расширяет формулу пути φ, либо вычисляет новую формулу абстракции ψ и сбрасывает φ. Точки вычисления абстракции (и, таким образом, размер блока) определяются так называемым оператором настройки блока *blk*. Если оператор $blk(e, g)$ возвращает $\bot$ (нет вычисления абстракции, т. е. абстрактное состояние e без абстракции), то следующее состояние $\langle \psi', \varphi' \rangle$ содержит $\psi' = \psi$ (неизмененное) и $\varphi' = SP_{op}(\varphi)$. Если оператор $blk(e, g)$ возвращает $\top$ (e — состояние абстракции), то следую-

щее состояние $\langle \psi', \varphi' \rangle$ содержит результат вычисления абстракции по формуле $\psi \wedge \varphi$, $\psi' = (SP_{op}(\varphi \wedge \psi))^{\pi}$ и $\varphi' = \top$ как новую дизъюнктивную формулу. Если $\varphi \wedge \psi$ невыполнимо для состояния e, то e недостижимо.

3.2.2. Модель памяти на основе неинтерпретируемых функций

Для моделирования памяти будем использовать неинтерпретируемые функции f (f_m) и B. Имя неинтерпретируемой функции f_m определим как конкатенацию имени f и индекса m: $f\#m$. Над формулами ψ для неинтерпретируемых функций f и f_m определим операцию подмены всех таких функций, содержащихся в формуле ψ на новое имя $f_{m'}$: $\psi[f_{m'}]$ — заменяет все вхождения f и f_m на $f_{m'}$.

Для задания модели памяти нам потребуются вспомогательные компоненты, которые будут храниться в абстрактном состоянии и изменяться при переходе к следующему состоянию. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times \texttt{int}$, где константа $A \in \mathcal{A}$ представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k – последний индекс базовых адресов.

В модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется таблицей 3.1.

Будем использовать вспомогательную функцию

$$mem\_upd(p, m', m, addrs) = \bigwedge_{(a,i) \in addrs} p = a + i \vee f_{m'}(a + i) = f_m(a + i),$$

где e — это выражение без побочных эффектов и без разыменований указателей.

В выражениях `*(s1 + i)`, `i` имеет тип `int`.

Размер целого и указателей принимается равным 1 байту.

Иначе нужно в выражении `*(s + i)` в `i` указывать размер как количество элементов, помноженное на соответствующий размер элемента.

3.2.3. Расширение подхода для структур

Подход к построению ограничений Γ может быть легко расширен на случай использования в программе структурных типов. Основная идея состоит в том, что структура рассматривается как участок памяти размера, равного сумме размеров полей. Обращения к полям транслируются как сумма указателя на начало структуры и смещения поля относительно начала. Обозначим как $\omega(A, f)$ смещение поля с именем f в структуре A.

1. При выделении памяти `alloc(size)` для структуры A, размер *size* определяется по размеру структуры — сумма размеров полей, и, возможно, с учетом выравнивания.

2. Выражение доступа к полю `s->f` рассматривается как `*(s +` $\omega(A, f)$`)`.

3. В присваивании структур по значению `s1 = s2` необходимо выписывать ограничения для присваивания всех полей структуры.

3.2.4. Пример построения формулы пути

В примере программы используется макрос `container_of(p, type, field_name)`, который раскрывается как `(type)(p + (0 -` ω`(type,field_name)))`. То есть для указателя на вложенное поле структуры `field_name`, мы получаем указатель на структуру, содержащую это поле. Макрос `container_of` часто используется в коде ядра ОС Linux и составляет большую сложность для инструментов, не поддерживающих адресную арифметику. Из-за этого возникают как ложные срабатывания, так и упущенные ошибки.

Операция (op)	Индекс функции памяти m'	Множество адресных переменных $Alloc'$	Индекс базового адреса k'	Ограничения Γ
Выделение переменной на стеке `int s;` или `int *s;`	Не меняется	A' — новое имя переменной $Alloc' = Alloc \cup \{(A', 0)\}$	k' — новый индекс	$s = A' \wedge$ $A' > 0 \wedge$ $B(A') = k'$
Выделение памяти размера `size` в куче `s = alloc(size)`	m' — новый индекс	A' — новое имя переменной $Alloc' = Alloc \cup \{(A', 0) \dots (A', \texttt{size} - 1)\}$	k' — новый индекс	$f_{m'}(s) = A' \wedge$ $A' > 0 \wedge$ $\bigwedge_{i=0}^{\texttt{size}-1} B(A' + i) = k' \wedge$, $mem\_upd(s, m', m, Alloc)$
`s = e`	m' — новый индекс	Не меняется	Не меняется	$f_{m'}(s) = \Gamma(\texttt{e}) \wedge$ $mem\_upd(s, m', m, Alloc)$, где $\Gamma(\texttt{e})$ для выражения `e` вычисляется по следующим правилам: $\Gamma(\texttt{const}) : const$ $\Gamma(\texttt{s}) : f_m(s)$ $\Gamma(\texttt{s1}\ op\ \texttt{s2})$, $op \in \{+, -, *, /\} :$ $f_m(s1)\ op\ f_m(s2)$
`*(s1 + i) = s2`	m' — новый индекс	Не меняется	Не меняется	$f_{m'}(f_m(s1) + f_m(i)) = f_m(s2) \wedge$ $mem\_upd(f_m(s1) + f_m(i), m', m, Alloc)$
`s1 = *(s2 + i)`	m' — новый индекс	Не меняется	Не меняется	$f_{m'}(s1) =$ $f_m(f_m(s2) + f_m(i)) \wedge$ $mem\_upd(s1, m', m, Alloc)$
`assume p`	Не меняется	Не меняется	Не меняется	$\Gamma(\texttt{p})$ для предиката `p` вычисляется по следующим правилам: $\Gamma(\texttt{const}) : const$ $\Gamma(\texttt{s}) : f_m(s)$ $\Gamma(\texttt{p1 == p2}) : \Gamma(p1) = \Gamma(p2)$ $\Gamma(\texttt{p1 < p2}) : \Gamma(p1) < \Gamma(p2)$ $\Gamma(\texttt{p1 <= p2}) : \Gamma(p1) \leq \Gamma(p2)$ $\Gamma(\texttt{p1 \|\| p2}) : \Gamma(p1) \vee \Gamma(p2)$ $\Gamma(\texttt{p1 \&\& p2}) : \Gamma(p1) \wedge \Gamma(p2)$ $\Gamma(\texttt{!p}) : \neg\Gamma(p)$

Таблица 3.1. Правила построения ограничений Γ.

Пусть задана следующая программа:

```
struct B { int a; int b; };
struct B *p; struct B *q; int *x;
p = alloc(sizeof(B));
p->b = 1;
x = &(p->a);
q = container\_of(x, struct B, a);
assume(p->b != q->b);
```

Программа содержит единственный путь, являющийся недостижимым, так как `p->b == q->b` и условие в последнем `assume` не выполнено. В данном примере при моделировании памяти требуется учитывать семантику арифметики указателей, иначе возможно ложное срабатывание из-за того, что путь может быть признан достижимым.

Для данного пути мы предварительно заменим операции со структурами на операции с указателями и построим для него формулу пути (см. табл. 3.2).

Таким образом, мы получим формулу пути $SP_\sigma(\top)$, являющуюся конъюнкцией формул в столбце Г. Эта формула является невыполнимой, что подтверждает недостижимость данного пути, что и требовалось показать в приведенном примере.

3.2.5. Конфигурируемый анализ (CPA) с моделью памяти на основе неинтерпретируемых функций

Формализуем анализ с моделью памяти на основе неинтерпретируемых функций в виде *конфигурируемого анализа* (Configurable program analysis, CPA) [137]. Это позволяет использовать гибкость операторов CPA для описания анализа без изменения основного алгоритма (см. Алгоритм 1).

Конфигурируемый анализ $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$ состоит из абстрактного домена D, отношения перехода $\rightsquigarrow$, оператора $merge$ и оператора $stop$,

Путь	Построение формулы пути			
	m'	$Alloc'$	k'	Γ
`struct B *p;`	0	$\{(A_0, 0)\}$	1	$p = A_0 \wedge$ $A_0 > 0 \wedge$ $B(A_0) = 1$
`struct B *q;`	0	$\{(A_0,0),(A_1,0)\}$	2	$q = A_1 \wedge$ $A_1 > 0 \wedge$ $B(A_1) = 2$
`int *x;`	0	$\{(A_0,0),(A_1,0),$ $(A_2,0)\}$	3	$x = A_2 \wedge$ $A_2 > 0 \wedge$ $B(A_2) = 3$
`p = alloc(` `sizeof(struct B));`	1	$\{(A_0,0),(A_1,0),$ $(A_2,0),(A_3,0),$ $...,$ $(A_3,$`sizeof(struct B)`$-1)\}$	4	$f_1(p) = A_3 \wedge$ $A_3 > 0 \wedge$ $\bigwedge_{i=0}^{\texttt{sizeof(struct B)}-1} B(A_3 + i) = 4 \wedge$ $\bigwedge_{(A_i,o_i)\in Alloc} \big(p = A_i + o_i \vee$ $f_1(A_i + o_i) = f_0(A_i + o_i)\big)$
`*(p +` $\omega(B, b)$`) = 1;`	2	$\{(A_0,0),(A_1,0),$ $(A_2,0),(A_3,0),$ $...,$ $(A_3,$`sizeof(struct B)`$-1)\}$	4	$f_2(f_1(p) + \omega(B, b)) = 1 \wedge$ $\bigwedge_{(A_i,o_i)\in Alloc} \big(f_1(p) + \omega(B, b) = A_i + o_i \vee$ $f_2(A_i + o_i) = f_1(A_i + o_i)\big)$
`x = p +` $\omega(B, a)$`;`	3	$\{(A_0,0),(A_1,0),$ $(A_2,0),(A_3,0),$ $...,$ $(A_3,$`sizeof(struct B)`$-1)\}$	4	$f_3(x) = f_2(p) + \omega(B, a) \wedge$ $\bigwedge_{(A_i,o_i)\in Alloc} \big(x = A_i + o_i \vee$ $f_3(A_i + o_i) = f_2(A_i + o_i)\big)$
`q = x +` `(0 -` $\omega(B, a)$`);`	4	$\{(A_0,0),(A_1,0),$ $(A_2,0),(A_3,0),$ $...,$ $(A_3,$`sizeof(struct B)`$-1)\}$	4	$f_4(q) = f_3(x) - \omega(B, a) \wedge$ $\bigwedge_{(A_i,o_i)\in Alloc} \big(q = A_i + o_i \vee$ $f_4(A_i + o_i) = f_3(A_i + o_i)\big)$
`assume` `*(p +` $\omega(B, b)$`) !=` `*(q +` $\omega(B, b)$`)`	4	$\{(A_0,0),(A_1,0),$ $(A_2,0),(A_3,0),$ $...,$ $(A_3,$`sizeof(struct B)`$-1)\}$	4	$\neg\Big(f_4\big(f_4(p) + \omega(B, b)\big) =$ $f_4\big(f_4(q) + \omega(B, b)\big)\Big)$

Таблица 3.2. Пример построения формулы пути.

определяемых следующим образом.

Пусть задана программа $P = (A, l_0, l_E)$, где X — обозначает множество переменных используемых в программе P, F — множество неинтерпретируемых функций, используемых для моделирования памяти, $\mathcal{P}$ — множество предикатов без кванторов над переменными X и функциями F, и $\Pi : L \to 2^{\mathcal{P}}$ — точность предикатной абстракции.

Абстрактный домен

Абстрактный домен $D = (C, R, [\![\cdot]\!])$ — это тройка, состоящая из множества конкретных состояний C, полурешетки $R = (E, \top, \sqsubseteq, \sqcup)$ и функции конкретизации $[\![\cdot]\!] : E \to C$.

Элементы решетки также называются абстрактными состояниями и являются семерками $(l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, где первые четыре компонента являются стандартными компонентами анализа ABE, $l, l^{\psi} \in L \cup \{l_{\top}\}$; $\psi, \varphi \in \mathcal{P}$. Компонент l моделирует счетчик команд, формула абстракции ψ — булева комбинация предикатов, заданных в Π, l^{ψ} — точка в программе, в которой была вычислена абстракция ψ, а φ — это дизъюнктивная формула, представляющая некоторые или все пути из точки l^{ψ} в l. Заметим, что в состоянии абстракции всегда $l = l^{\psi}$ и $\varphi = \top$.

Для предложенной модели памяти мы ввели три новых компонента. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times \texttt{int}$, где $A \in \mathcal{A}$ — представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k — последний индекс базовых адресов.

Верхний элемент решетки — это абстрактное состояние $\top = (l_{\top}, \top, l_{\top}, \top, \emptyset, 0, 0)$. Частичный порядок $\sqsubseteq \subseteq E \times E$ определяется так, что для любых двух состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 =$

$(l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ из E выполнено:

$$e_1 \sqsubseteq e_2 \equiv \big((e_2 = \ top) \vee (l_1 = l_2 \wedge \varphi_1 = \varphi_2 = \top \wedge \psi_1[f] \implies \psi_2[f])\big).$$

Заметим, что мы подменяем все вхождения версий памяти в формулах абстракции $\psi_1[f]$ и $\psi_2[f]$ на одинаковую версию f.

Оператор соединения $\sqcup : E \times E \to E$ выдает наименьшую верхнюю грань двух операндов в соответствии с частичным порядком $\sqsubseteq$.

Отношение перехода

Отношение перехода $\rightsquigarrow \subseteq E \times G \times E$ содержит все дуги (e, g, e'), где $e = (l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, $e' = (l', \psi', l^{\psi'}, \varphi', Alloc', k', m')$ и $g = (l, op, l')$, для которых выполнено:

$$\begin{cases} \varphi' = \top \wedge \psi' = \big(SP_{op}(\varphi \wedge \psi)\big)^{\Pi(l')}[f_{m'}] \wedge l^{\psi'} = l', & \text{если } blk(e, g) \vee (l' = l_E) \\ \varphi' = SP_{op}(\varphi) \wedge \psi' = \psi \wedge l^{\psi'} = l^{\psi}, & \text{иначе.} \end{cases}$$

В представленной модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется таблицей 3.1. При этом значения $Alloc', k', m'$ в следующем состоянии определяются также по таблице 3.1.

Таким образом, мы имеем отношения перехода, которое работает в двух режимах, определяемых оператором $blk : E \times G \to \mathbb{B}$, который отображает абстрактное состояние e и дугу g АПУ в $\top$ или $\bot$. Оператор blk задается как параметр анализу. В первом режиме строится абстракция, а во втором вычисляется только сильнейшее постусловие.

Оператор слияния

Оператор слияния $merge : E \times E \to E$ для двух абстрактных состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ опреде-

Алгоритм 1 Алгоритм конфигурируемого анализа (CPA) $(\mathbb{D}, e_0)$ (взят из работы [137]).

Вход: CPA $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$, начальное состояние $e_0 \in E$, где E обозначает множество элементов решетки D

Переменные: множество *reached* достигнутых элементов из E, множество *waitlist* элементов из E

Выход: множество достижимых абстрактных состояний

```
1  waitlist ← {e_0}
2  reached ← {e_0}
3  пока waitlist ≠ ∅ делать
4      выбрать элемент e из waitlist
5      waitlist ← waitlist \ {e}
6      для e' ∈ E таких, что e ⇝ e' делать
7          для e'' ∈ reached делать
                   ▷ слияние с существующими абстрактными состояниями
8              e_new ← merge(e', e'')
9              если e_new ≠ e'' то
10                 waitlist ← (waitlist ∪ {e_new}) \ {e''}
11                 reached ← (reached ∪ {e_new}) \ {e''}
12             если ¬stop(e', reached) то
13                 waitlist ← waitlist ∪ {e'}
14                 reached ← reached ∪ {e'}
15 вернуть reached
```

ляется следующим образом:

$$merge(e_1, e_2) = \begin{cases} \Big(l_2, \psi_2, l^{\psi_2}, \\ \quad \big(\varphi_1 \wedge mem\_unch(m', m_1, Alloc_1)\big) \vee \\ \quad\quad \big(\varphi_2 \wedge mem\_unch(m', m_2, Alloc_2)\big), \\ \quad Alloc_1 \cup Alloc_2, \max(k_1, k_2), m'\Big), & \text{если } l_1 = l_2 \wedge \psi_1[f] = \psi_2[f] \wedge l^{\psi_1} = l^{\psi_2} \\ e_2, & \text{иначе,} \end{cases}$$

где m' — новый индекс функции памяти, а функция mem_unch определяется как:

$$mem\_unch(m', m, addrs) = \bigwedge_{(a,i) \in addrs} f_{m'}(a + i) = f_m(a + i).$$

Таким образом, множества адресных переменных объединяются, а при построении новой дизъюнктивной формулы вводится новый индекс функции памяти, к которой приравниваются функции в каждом из состояний.

Оператор останова

Оператор останова $stop : E \times 2^E \to \mathbb{B}$ проверяет, покрывается ли состояние e другим состоянием из пройденных состояний R (множество *reached*):

$$\forall e \in E \,.\, \forall R \subseteq E \,.\, stop(e, R) = \exists e' \in R \,.\, e \sqsubseteq e'.$$

3.3. Оптимизации

Эффективность модели памяти сильно зависит от количества дизъюнкций, выписываемых в mem_upd из Таблицы 3.1. В реализации описанного подхода предлагается использовать следующие оптимизации для сокращения числа этих дизъюнкций:

1. Разделение области памяти по типам, так что каждая неинтерпретируемая функция задает отображение адресов переменных в их значения для одного соответствующего ей примитивного типа данных. Например, вводятся функции $\{f_{\texttt{long\_int}}, f_{\texttt{char *}}, f_{\texttt{struct B *}}\}$. Примитивными, то есть не составными типами данных в данном подходе, считаются символьный и целочисленный типы, а также любой тип указателя. Таким образом, неявно предполагается, что значение, записанное по какому-либо адресу в качестве значения какого-то типа данных не может быть впо-

следствии считано по этому же адресу как значение другого типа. Это предположение является одним из ограничений подхода.

2. Выделение среди множества всех переменных программы подмножества “чистых” переменных, к которым возможен доступ лишь по именам (то есть переменных, не имеющих алиасов). Для таких переменных нет необходимости в использовании неинтерпретируемых функций для представления значений. Поэтому для них используется сходное с используемым инструментами BLAST и CPAchecker SSA-представление (для них не выписывается разыменование и имя переменной представляет значение переменной, а не ее адрес). Определение “чистых” переменных возможно проводить не для всей программы, а на заданном пути, до тех пор, пока не встретиться взятие адреса для этой переменной на этом пути или на другом пути при выполнении оператора $merge$.

3. Использование эвристики для полей структур. В предлагаемом подходе предполагается, что указатель на поле структуры, получаемый сложением адреса структуры с соответствующим смещением поля, может принимать только значения адресов того же самого поля в других структурах того же самого типа, что и структура, которой это поле принадлежит. Такой указатель не может, в частности, быть равным адресу элемента массива или отдельной переменной, не являющейся полем структуры. Например, обновление поля `skb1->next` не может повлиять ни на какой `skb2->prev`, даже если `next` и `prev` одного типа. В этом случае в качестве оптимизации мы опускаем посыл импликации в mem_upd, если смещения заранее не равны. Такое предположение также является одним из ограничений метода.

4. Инициализация константами. Оптимизация заключается в том, что мы выписываем одно обновление *mem_ upd* для нескольких присваиваний. Например, при выделении памяти, заполненной нулями под структуру `kzalloc(sizeof(*info), GFP_KERNEL)`, *mem_ upd.* выписывается только один раз после инициализации всех полей нулями.

5. Сокращение множества *Alloc*, за счет сохранения смещений только для тех полей структур, которые были использованы в пути. Соответственно, для неиспользованных полей не будут выписываться дизъюнкции в *mem_ upd.*

3.4. Результаты

Предложенный метод был реализован в инструменте CPAchecker версии 1.4 (для экспериментов была взята ревизия `18237`). За включение разработанной модели памяти на основе неинтерпретируемых функций отвечает опция `cpa.predicate.handlePointerAliasing`. Эксперименты проводились на наборах тестовых программ SV-COMP'2016 [138].

В первую очередь была рассмотрена категория по работе со структурами данных в куче `Heap Data Structures`. Задачи в этой категории требуют поддержки анализа указателей.

Запуски для этой категории проводились в двух конфигурациях предикатного анализа `predicateAnalysis` с поддержкой модели памяти на основе неинтерпретируемых функций (с НФ) и без нее (без НФ). Как можно видеть в табл. 3.3, на представленном наборе анализ с использованием предложенного метода не дает некорректных результатов на представленном наборе по сравнению с не использующей его версией, которая давала 24 некорректных результата.

Отметим, что вердикт “unknown” возникает, когда верификатор не может найти ошибку или доказать ее отсутствие, в силу превышения лимитов по времени, памяти, обнаружению неподдерживаемых конструкций (например, рекурсии) или в силу ограничений самого CEGAR.

Количество вердиктов “unknown” составило 14 шт. для НФ, большая часть из них из-за превышения лимита по времени — 10 шт. В результате общее время работы анализа с НФ составило 9514 секунд, что почти в десять раз больше чем без НФ (см. табл. 3.4). Однако если рассматривать только время на корректные результаты (не включающее время на вердикты “unknown”), то время работы оказывается сравнимо.

Модели памяти	без НФ	с НФ
Общее количество	81	81
Корректные результаты за 900 с	53	67
Доказано отсутствие ошибки	34	44
Ошибка найдена	19	23
Некорректные результаты за 900 с	24	0
Упущенная ошибка	5	0
Ложное предупреждение	19	0
Завершено по истечении 900 с	4	14

Таблица 3.3. Результаты запуска на наборе `HeapDataStructures` в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

В качестве второго набора был выбран `DeviceDriversLinux64`, состоящий из драйверов устройств ядра операционной системы Linux, большая часть которого подготовлена в рамках проекта LDV [122, 139]. Для этого набора была использована конфигурация `ldv`, использующаяся в проекте LDV. Видно, что время работы для этого набора сравнимо как на всех тестах, так и

Модели памяти	без НФ	с НФ
Общее время	680	9514
Время для корректных результатов	476	487

Таблица 3.4. Время работы CPU (в секундах) запуска на наборе `HeapDataStructures` в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

на тестах, для которых получен корректный результат (см. табл. 3.6). Причем корректных результатов без НФ получено на 11 шт. больше (см. табл. 3.5).

Таблица 3.7 показывает изменения вердиктов при переходе от запуска без НФ к запуску с НФ. Видно, что модель памяти с НФ теряет 31 корректный результат, из них 29 из-за вердикта "`unknown`"("анализ не завершен"), одно ложное срабатывание получено из-за того, что для точной работы модели памяти с НФ в тесте не хватает явного выделения памяти, кроме того, еще в одном тесте анализ без НФ находит ложную трассу ошибки, так как считает возможным равенство нулю адреса переменной на стеке. С другой стороны, получается 20 новых корректных результатов, а 5 некорректных результатов становятся "`unknown`".

В рамках дипломной работы [140] студентом университета Пассау (Германия), в котором работает основная команда разработчиков инструмента CPAchecker было также произведено сравнение результатов верификации для трех моделей памяти: предложенной в данной работе модели на основе неинтерпретируемых функций без кванторов, модели с неинтерпретируемыми функциями в логике первого порядка и модели на основе теории массивов на наборе `DeviceDriversLinux64`. Модель, использующая логику первого порядка, и модель на основе массивов по сути являются частными случаями типизированной модели памяти, рассмотренной в главе 1. Интерполяция Крейга для этих моделей памяти не замкнута относительно используемых

теорий (для массивов), либо не полностью поддерживается решателями в общем случае (для логики первого порядка). Однако соответствующие практические результаты, приведенные в таблицах 3.8 и 3.9, показывают, что прирост корректных вердиктов в предложенной в данной работе модели памяти по сравнению с моделью, использующей теорию массивов, примерно равен приросту при сравнении модели, использующей теорию массивов и модели, использующей логику первого порядка, и в зависимости от используемого интерполирующего решателя варьируется в пределах от нескольких единиц (6 для решателя Z3) до нескольких десятков (92 для решателя MathSAT5).

Модели памяти	без НФ	с НФ
Общее количество	2121	2121
Корректные результаты за 900 с	1654	1643
Доказано отсутствие ошибки	1450	1450
Ошибка найдена	204	193
Некорректные результаты за 900 с	17	6
Упущенная ошибка	5	3
Ложное предупреждение	12	3
Завершено по истечении 900 с	450	472

Таблица 3.5. Результаты запуска на наборе `DeviceDriversLinux64` в конфигурации `ldv`, с лимитом по времени 15 минут и 15 Gb памяти.

Модели памяти	без НФ	с НФ
Общее время	140 часов	144 часа
Время для корректных результатов	22,2 часов	22,7 часов

Таблица 3.6. Время работы CPU (в часах) запуска на наборе `DeviceDriversLinux64` в конфигурации `ldv`, с лимитом по времени 15 минут и 15 Gb по памяти.

Переходы из модели памяти без НФ в модель памяти с НФ			Количество
Корректный результат	→	Некорректный результат	2
Корректный результат	→	Анализ не завершен	29
Некорректный результат	→	Корректный результат	8
Некорректный результат	→	Анализ не завершен	5
Анализ не завершен	→	Корректный результат	12
Анализ не завершен	→	Некорректный результат	0

Таблица 3.7. Результаты изменения вердиктов на наборе `DeviceDriversLinux64` в конфигурации `ldv`.

3.5. Выводы

Предложенный в данной главе метод моделирования памяти на основе неинтерпретируемых функций позволяет анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы, и выражения, содержащие адресную арифметику. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Метод является масштабируемым, так как показывает приемлемые результаты по скорости на практически значимом наборе из драйверов устройств ОС Linux. Вместе с тем, ряд корректных результатов не может быть получен, по причине замедления работы анализа. Это требует дальнейшего развития метода.

Метод обладает большей производительностью по сравнению с методами, использующими логику первого порядка или теорию массивов и не накладывающими ограничение на размеры моделируемых областей памяти.

Среди возможных направлений отметим, возможность более точного разбиения на регионы памяти, так что для этих регионов выделяются независимые неинтерпретируемые функции. Например, отдельные функции могут

Модели памяти	массивы НФ с кванторами **НФ без кванторов**			
SMT-решатели	MathSAT5	Princess	SMTInterpol	Z3
Общее количество	2120	2120	2120	2120
Корректные результаты за 900 с	1173	1133	1303	1301
	—	1040	—	1293
	1265	**1198**	**1326**	**1306**
Отсутствие ошибки	1126	1091	1218	1217
	—	1006	—	1217
	1197	**1143**	**1230**	**1222**
Ошибка найдена	47	42	85	**84**
	—	34	—	76
	68	**55**	**96**	**84**
Некорректные результаты за 900 с	**2**	**1**	**3**	3
	—	0	—	6
	3	**1**	**3**	**2**
Упущенная ошибка	**0**	**0**	**0**	**0**
	—	**0**	—	2
	0	**0**	**0**	**0**
Ложное предупреждение	**2**	**1**	**3**	3
	—	**0**	—	4
	3	**1**	**3**	**2**
Завершено по истечении 900 с	945	986	814	816
	—	1080	—	821
	852	**921**	**791**	**812**

Таблица 3.8. Результаты запуска на наборе `DeviceDriversLinux64` в конфигурации `predicateAnalysis`, с лимитом по времени 15 минут и 15 Gb памяти.

Модели памяти	массивы НФ и кванторы **НФ без кванторов**			
SMT-решатели	MathSAT5	Princess	SMTInterpol	Z3
Общее время	451	623	**276**	332
	—	**477**	—	349
	374	523	309	**313**
Время для корректных результатов	**21**	41	**32**	30
	—	**32**	—	31
	30	**38**	36	**25**

Таблица 3.9. Время работы CPU (в тысячах секунд) запуска на наборе `DeviceDriversLinux64` в конфигурации `predicateAnalysis`, с лимитом по времени 15 минут и 15 Gb по памяти.

быть использованы для каждого поля структуры, для которого не берется адрес. Регионы также могут быть выделены на основе предварительного анализа кода программы. Кроме того, в качестве направления исследований может быть рассмотрено объединение инструкций программы, например, объединение последовательности присваиваний, меняющих независимые участки памяти, и их рассмотрение как одного обновления памяти.

Реализация модели на основе теории неинтерпретируемых функций в инструменте статической верификации CPAchecker позволила увеличить точность предикатного анализа в инструменте без значительных потерь в его эффективности, а также уменьшить число ложных сообщений об ошибках при верификации модулей ядра ОС Linux в рамках проекта LDV и верифицировать модули ядра относительно новых правил корректности, существенно использующих адресную арифметику. Реализация предложенной модели

памяти допускает повторное использование для различных видов анализа программ, отличных от предикатной абстракции с уточнением. Из поддерживаемых инструментом видов статического анализа соответствующая реализация на момент написания работы использовалась по крайней мере еще в трех: ограничиваемой верификации (BMC), k-индукции и верификации с помощью алгоритма Impact [141].

Глава 4

Модель памяти с вложенными регионами для дедуктивной верификации

В данной главе описывается новый вид анализа регионов для дедуктивной верификации Си-программ на основе базового языка с поддержкой произвольной вложенности структур, объединений и массивов в другие структуры. В разделе 4.1 вводится базовый язык с помощью описания его абстрактного синтаксиса, правил типизации и исходной семантики. Далее в разделе 4.2 предлагаются нормализующие преобразования, позволяющие преобразовать исходную аннотированную Си-программу, содержащую произвольные случаи использования объединений, префиксных и непрефиксных приведений типов указателей на структуры, а также конструкции для получения указателя на объемлющую структуру, в соответствующую программу на новом базовом языке (нормализация описывается для последовательностей операторов). В разделе 4.3 приводится модельная семантика базового языка и доказываются теоремы о корректности и полноте этой семантики относительно исходной. В разделе 4.4 приводятся дополнительные ограничения на контекст типизации термов базового языка, определяющие результат анализа регионов для базового языка, а также описывается набор ограничений на входные программы, при которых предлагаемые ограничения на контекст обеспечивают полное моделирование семантики входных программ. Приводится схема доказательства соответствующей теоремы о полноте. В этом разделе также кратко обсуждаются проблемы и решения, связанные с поддержкой контекстно-зависимого анализа регионов для вызовов функций, необходимого для применения предложенной модели памяти с регионами при верификации реального Си-кода. В заключении обсуждаются ограничения нового анализа регионов при

применении его для верификации произвольных программ на языке Си.

4.1. Базовый язык с поддержкой вложенности

4.1.1. Абстрактный синтаксис и типизация

Абстрактный синтаксис [77] базового языка и соответствующие правила типизации приведены на рис. 4.1. Рассматриваемый базовый язык позволяет выражать только конечные последовательности следующих видов инструкций (в абстрактном синтаксисе обозначены символами o и o'):

- инструкции считывания и обновления значений переменных и адресуемых указателями ячеек памяти ($v = t_v$, $p = t_p$, $p\text{->}f_v = v$, $p_1\text{->}f_p = p_2$),
- инструкции проверки ($\mathbf{assert}(v \lozenge 0)$) и предположения ($\mathbf{assume}(v \lozenge 0)$) утверждений о значениях переменных,
- инструкции динамического выделения и освобождения памяти ($p = \mathbf{alloc}(v)$ и $\mathbf{free}(p)$), а также
- три вида специальных спецификационных инструкций ($\mathbf{to\_int}(p, v)$, $\mathbf{of\_int}(p, s, v)$ и $\mathbf{may\_alias}(p_1, p_2)$), введенных для обеспечения полноты модельной семантики базового языка относительно его исходной семантики.

Такое ограниченное в смысле набора конструкций рассмотрение базового языка не ограничивает его применения в качестве основы для модели памяти в реальном инструменте дедуктивной верификации, так как соответствующее расширение этого языка условными операторами, циклами и поддержкой определений и вызовов функций может быть осуществлено с использованием широко известных техник [49, 142] (см. исчисление слабейших предусловий для соответствующих конструкций).

Для работы с указателями (на структуры) в базовый язык включены конструкции, соответствующие разности указателей ($p_1 - p_2$), проверке равенства указателей ($p_1 == p_2$), сдвигу ($p + v$) и разыменованию указателя ($p\texttt{->}f_v$ и $f\texttt{->}f_p$), а также конструкции для получения указателя на вложенную ($\texttt{\&}p\texttt{->}f_s$) и объемлющую ($\textbf{container\_of}(p, f_{s'})$) структуры. Конструкция $\textbf{container\_of}(p, f_{s'})$ аналогична одноименному макросу, используемому в ядре Linux (объявлен в файле `include/linux/kernel.h`).

Предполагается, что для последовательностей инструкций базового языка (обозначены символом oo) задан соответствующий контекст, состоящий из множеств $\mathbb{S}$, $\mathbb{F}_v$, $\mathbb{F}_p$, $\mathbb{F}_s$, $\mathbb{P}$, $\mathbb{V}$ и $\mathbb{R}$, отображения $\varphi : \mathbb{S} \to 2^{\mathbb{F}_v \cup \mathbb{F}_p \cup \mathbb{F}_s}$ и контекстов типизации Φ и Γ, задающих соответственно типы для элементов множества $\mathbb{F} = \mathbb{F}_v \cup \mathbb{F}_p \cup \mathbb{F}_s$ и множества $\mathbb{V} \cup \mathbb{P} \cup \mathcal{T}^*$, где $\mathcal{T}^*$ — замыкание множества термов базового языка $\mathcal{T}$ относительно операции подстановки указательного терма t_{p_2} вместо указательной переменной p того же типа в некотором указательном терме t_{p_1}. $\mathbb{S}$ — это множество уникальных тегов структур, $\mathbb{F}_v$ — уникальных имен целочисленных полей, $\mathbb{F}_p$ — указательных полей, $\mathbb{F}_s$ — полей-массивов структурного типа, $\mathbb{V}$ — множество целочисленных переменных, $\mathbb{P}$ — указательных переменных, $\mathbb{R}$ — множество имен регионов для типизации указательных термов, отображение φ задает множество имен полей для каждой структуры с тегом из множества $\mathbb{S}$. Множества имен полей различных структур не пересекаются. Контекст Φ типизации полей структур отображает поля из множества $\mathbb{F}_v$ в тип `int`, поля из множества $\mathbb{F}_p$ — в типы вида `pointer` s для некоторых тегов структур s из $\mathbb{S}$, а поля из множества $\mathbb{F}_s$ — в типы вида `struct` s `[`c`]`, где $s \in \mathbb{S}$, а константа $c \in \mathbb{N}\backslash\{0\}$, где 0 считается принадлежащим множеству натуральных чисел $\mathbb{N}$. Контекст типизации Γ отображает целочисленные переменные из множества $\mathbb{V}$ в тип `int`, указательные переменные из множества $\mathbb{P}$ в типы вида `pointer` $s\,\rho$, $s \in \mathbb{S}$, $\rho \in \mathbb{R}$, а также термы базового языка в их типы, задаваемые правилами вывода, приведенными на

$$\{\text{int}, \text{void}\} \subseteq \mathbb{S}, \{f_{s.\text{void}} | s \in \mathbb{S}\} \subseteq \mathbb{F}_s, f_{\text{int.int}} \in \mathbb{F}_v,$$

$$s, s' \in \mathbb{S}, \varphi(s) \subseteq 2^{\mathbb{F}_v \cup \mathbb{F}_p \cup \mathbb{F}_s}, s \neq s' \Rightarrow \varphi(s) \cap \varphi(s') = \emptyset,$$

$$\varphi(\text{int}) = \{f_{\text{int.int}}\}, \varphi(\text{void}) = \emptyset, s \neq \text{void} \Rightarrow f_{s.\text{void}} \in \varphi(s),$$

$$\rho, \rho' \in \mathbb{R}, \Diamond \in \{<, >, =, \neq, \leq, \geq\}, c \in \mathbb{N} \setminus \{0\}, \mathbb{B} = \{0, 1\},$$

$$f_v : \texttt{int} \in \mathbb{F}_v, f_p : \texttt{pointer}\ s \in \mathbb{F}_p, f_s : \texttt{struct}\ s\ [c] \in \mathbb{F}_s,$$

$$f_{\text{int.int}} : \texttt{int}, f_{s.\text{void}} : \texttt{struct}\ \text{void}\ [1], i \in \mathbb{A}_{\mathbb{V},\mathbb{B}^d}, p, p_1, p_2 \in \mathbb{P}, v \in \mathbb{V}$$

$t_v ::=$		$(t_v : \texttt{int})$
	i	$i : \texttt{int}$
	$p_1 - p_2$	$p_1, p_2 : \texttt{pointer}\ s\ \rho$
	$p_1 == p_2$	$p_1, p_2 : \texttt{pointer}\ s\ \rho$
	$p\texttt{->}f_v$	$p : \texttt{pointer}\ s\ \rho, f_v \in \varphi(s)$
$t_p ::=$		$(t_p : \texttt{pointer}\ s\ \rho)$
	NULL	
	$p + v$	$p : \texttt{pointer}\ s\ \rho, v : \texttt{int}$
	$p\texttt{->}f_p$	$p : \texttt{pointer}\ s'\ \rho', f_p : \texttt{pointer}\ s\ \in \varphi(s')$
	$\texttt{\&}p\texttt{->}f_s$	$p : \texttt{pointer}\ s'\ \rho', f_s : \texttt{struct}\ s\ [c] \in \varphi(s')$
	$\textbf{container\_of}(p, f_{s'})$	$p : \texttt{pointer}\ s'\ \rho', f_{s'} : \texttt{struct}\ s'\ [c] \in \varphi(s)$
$o ::=$		$(o : \texttt{unit})$
	$v = t_v$	$v, t_v : \texttt{int}$
	$p = t_p$	$p, t_p : \texttt{pointer}\ s\ \rho$
	$p = \textbf{alloc}(v)$	$p : \texttt{pointer}\ s\ \rho, v : \texttt{int}, s \neq \text{void}$
	$\textbf{free}(p)$	$p : \texttt{pointer}\ s\ \rho, s \neq \text{void}$
	$p\texttt{->}f_v = v$	$p : \texttt{pointer}\ s\ \rho, f_v \in \varphi(s), v : \texttt{int}$
	$p_1\texttt{->}f_p = p_2$	$p_1 : \texttt{pointer}\ s\ \rho, f_p : \texttt{pointer}\ s'\ \in \varphi(s), p_2 : \texttt{pointer}\ s'\ \rho'$
	$\textbf{assert}(v \Diamond 0)$	$v : \texttt{int}$
	$\textbf{assume}(v \Diamond 0)$	$v : \texttt{int}$
$o' ::=$		$(o' : \texttt{unit})$
	$\textbf{to\_int}(p, v)$	$p : \texttt{pointer}\ s\ \rho, v : \texttt{int}$
	$\textbf{of\_int}(p, s, v)$	$p : \texttt{pointer int}\ \rho, v : \texttt{int}$
	$\textbf{may\_alias}(p_1, p_2)$	$p_1, p_2 : \texttt{pointer}\ s\ \rho$
$oo ::=$		$(oo : \texttt{unit})$
	ε	
	$o; oo$	$o, oo : \texttt{unit}$
	$o'; oo$	$o', oo : \texttt{unit}$

Рис. 4.1. Абстрактный синтаксис и типизация термов базового языка

рис. 4.1. Правила вывода для компактности размещены рядом с правилами

построения термов и читаются следующим образом: для правила вида

$$\begin{array}{rll} t ::= & & (C) \\ & \mid \ldots & \ldots \\ & \mid D & A_1, A_2, \ldots A_n \end{array}$$

соответствующее правило вывода типа —

$$\frac{\Gamma \vdash A_1, A_2, \ldots, A_n}{\Gamma \vdash [D/t]C}.$$

Правило вывода типа для терма вида $p = t_p$ требует одинакового тега структуры и имени региона для указательной переменной p и терма t_p. Это позволяет по приведенным правилам типизации корректно (но не однозначно) определить типы не только для термов базового языка, но и для замыкания $\mathcal{T}^*$ множества термов базового языка $\mathcal{T}$ относительно операции подстановки указательного терма t_{p_2} вместо указательной переменной p того же типа в любом указательном терме t_{p_2}. Для этого достаточно вместо каждой подстановки вида $[t_{p_2}/p]t_{p_1}$ рассмотреть пару инструкций $p' = t_{p_2}; p'' = [p'/p]t_{p_1}$ и тогда $\Gamma([t_{p_2}/p]t_{p_1}) = \Gamma(p'')$. Таким образом, контекст типизации Γ является отображением $\mathbb{V} \cup \mathbb{P} \cup \mathcal{T}^* \to \{\texttt{int}, \texttt{unit}\} \cup \{\texttt{pointer}\, s\, \rho \big| s \in \mathbb{S}, \rho \in \mathbb{R}\}$.

Предполагается, что множество $\mathbb{S}$ всегда содержит по крайней мере два тега для структур int и void, множество $\mathbb{F}_v$ — хотя бы одно имя поля $f_{\text{int.int}}$, множество $\mathbb{F}_s$ — имена вида $f_{s.\text{void}}$ для всех структур $s \neq \text{void}$, причем $\varphi(\text{int}) = \{f_{\text{int.int}}\}$, а $\varphi(\text{void}) = \emptyset$. Поле $f_{\text{int.int}}$ имеет тип `int`, а поля $f_{s.\text{void}}$ — тип `struct` void `[1]`. Метапеременная [77] i обозначает арифметический терм над переменными из множества $\mathbb{V}$ и константами из множества $\mathbb{B}^d$ битовых векторов длины d (соответствующее множество термов на рис. 4.1 обозначено $\mathbb{A}_{\mathbb{V},\mathbb{B}^d}$). Конкретный синтаксис и семантика арифметических термов не рассматриваются в данной работе, но предполагается, что по крайней мере для переменных и констант в арифметических термах доступны операции

сложения, умножения и вычитания по модулю 2^d (с точки зрения семантики одновременно знаковые и беззнаковые для представления в дополнительном коде), а также операция целочисленного знакового деления (с условием $-2^d \div -1 = -2^d$). Символом $\Diamond$ на рис. 4.1 обозначается одно из отношений $<, >, =, \neq, \leq, \geq$.

4.1.2. Исходная семантика

Для задания исходной семантики базового языка сделаем некоторые дополнительные предположения и введем необходимые обозначения, представленные на рис. 4.2.

На каждом множестве полей структур $\varphi(s)$ введем порядковую нумерацию полей с помощью конечных наборов множеств префиксов $\varphi_j(s)$, $j \in \mathbb{N}$, $0 \leq j \leq |\varphi(s)|$. Каждый нулевой префикс $\varphi_0(s)$ пуст для любого $s \in \mathbb{S}$, первый префикс для структур с тегом $s \neq \text{void}$ равен $\{f_{s.\text{void}}\}$, последний префикс $\varphi_{|\varphi(s)|}(s)$ совпадает со всем множеством полей $\varphi(s)$ для любого $s \in \mathbb{S}$. Каждый следующий префикс отличается от предыдущего ровно на одно поле. Для полей из множества $\mathbb{F}$ и структур из множества $\mathbb{S}$ введем отображение $\sigma : \mathbb{S} \cup \mathbb{F} \to \mathbb{N}$, задающее размер соответствующих полей или структур. Для корректного определения размера никакая структура с тегом s не должна транзитивно включать поле-массив структур с тем же тегом. Для введения этого ограничения, а также в целях дальнейшего использования при описании модельной семантики введем функцию $\pi(s, p, a) : \mathbb{S} \times \mathbb{T} \times \mathbb{N} \to 2^{\mathbb{S}\times\mathbb{T}\times\mathbb{N}}$, отображающую тройки вида (s, p, a), где s — тег структуры, a — натуральное число, используемое в качестве адреса, p — терм вида $\texttt{\&}(\texttt{\&}(\ldots(\texttt{\&}p\texttt{->}f_{s1})\ldots)\texttt{->}f_{sn-1})\texttt{->}f_{sn}$, $n \geq 0$ из подмножества $\mathbb{T} \subseteq \mathcal{T}^*$, в которое входят только термы указанного вида, во множества таких же троек для всех структур, транзитивно вложенных в структуру с тегом s

$$\forall j \in \mathbb{N}\mathbf{.}\ 1 \le j \le |\varphi(s)| \implies \varphi_j(s) \subseteq \varphi(s) \wedge \varphi_{j-1} \subset \varphi_j \wedge |\varphi_j \backslash \varphi_{j-1}| = 1,$$

$$\forall s \in \mathbb{S}\mathbf{.}\ s \ne \text{void} \implies \varphi_1(s) = \{f_{s.\text{void}}\}, \mathbb{F} = \mathbb{F}_v \cup \mathbb{F}_p \cup \mathbb{F}_s, \sigma : \mathbb{S} \cup \mathbb{F} \to \mathbb{N}, o : \mathbb{F} \to \mathbb{N},$$

$$\mathbb{T}_0 = \mathbb{P}, \forall i \in \mathbb{N}\mathbf{.}\ \mathbb{T}_{i+1} = \{\&p\texttt{->}f_s : \texttt{pointer}\, s\, \rho \big| p : \texttt{pointer}\, s'\, \rho' \in \mathbb{T}_i, f_s \in \varphi(s') \cap \mathbb{F}_s\},$$

$$\mathbb{T} = \bigcup_{i=0}^{\infty} \mathbb{T}_i, \pi : \mathbb{S} \times \mathbb{T} \times \mathbb{N} \to 2^{\mathbb{S}\times\mathbb{T}\times\mathbb{N}},$$

$$\pi(s,p,a) = \{(s,p,a)\} \cup \bigcup_{\substack{f_{s'}:\texttt{struct}\, s'\, [c] \\ \in \varphi(s) \cap \mathbb{F}_s}} \bigcup_{i=0}^{c-1} \pi\big(s', \&p\texttt{->}f_{s'}, a + o(f_{s'}) + \sigma(s') \times i\big),$$

$$s' \sqsubset^+ s \Leftrightarrow \big(\exists p : \texttt{pointer}\, s\, \rho \in \mathbb{P} \Rightarrow \exists p' \in \mathbb{T}, a' \in \mathbb{N}\mathbf{.}\ p' \ne p \wedge (s', p', a') \in \pi(s, p, 0)\big),$$

$$\forall s \in \mathbb{S}\mathbf{.}\ s \not\sqsubset^+ s, \sigma(s) = \textstyle\sum_{f \in \varphi(s)} \sigma(f),$$

$$\sigma(f_v) = \sigma(f_p) = 1, \sigma(f_s : \texttt{struct}\, s\, \texttt{[}c\texttt{]}) = \sigma(s) \times c, o(f) = \textstyle\sum_{\substack{f' \in \varphi_{j-1}(s) \\ f \in \varphi_j(s) \\ f \notin \varphi_{j-1}(s)}} \sigma(f'),$$

$$\forall x = x_{d-1} \ldots x_0 \in \mathbb{B}^d\mathbf{.}\ \langle x \rangle_U = \textstyle\sum_{j=0}^{d-1} x_j \times 2^j \wedge \langle x \rangle_S = -x_{d-1} \times 2^d + \sum_{j=0}^{d-2} x_j \times 2^j,$$

$$\forall x \in \mathbb{Z}\mathbf{.}\ \widehat{x} \in \mathbb{B}^d \wedge \langle \widehat{x} \rangle_U \equiv x \pmod{2^d},$$

$$\forall x, y \in \mathbb{B}^d\mathbf{.}\ x \,\widehat{\star}\, y = \widehat{\langle x \rangle_U \star \langle y \rangle_U} \wedge x \,\widehat{\div}_S\, y = \widehat{\langle s \rangle_S \div \langle y \rangle_S} \wedge x \,\widehat{\Diamond}\, y = \langle x \rangle_S \Diamond \langle y \rangle_S, \star \in \{+, -, \times\},$$

$$B : \mathbb{B}^d \to \mathbb{B}^d, V : \mathbb{B}^d \to \mathbb{B}, L : \mathbb{B}^d \to \mathbb{B}^d, E_p : \mathbb{P} \to \mathbb{B}^d, E_v : \mathbb{V} \to \mathbb{B}^d, S = (B, V, L, E_p, E_v),$$

$$V_0 = \{a \mapsto 0 \big| a \in \mathbb{B}^d\}, L_0 = \{a \mapsto \widehat{0} \big| a \in \mathbb{B}^d\}, init(S) = V = V_0 \wedge L = L_0, 0 < l < u < 2^d$$

$$upd^v(S, v, i) = (B, V, L, E_p, E_v[v \leftarrow i])\ \ upd^p(S, p, a) = (B, V, L, E_p[p \leftarrow a], E_v)$$

$$upd^B(S, a, v) = (B[a \leftarrow v], V, L, E_p, E_v)$$

$$valid(S, p, f) = V\big[E_p[p] \,\widehat{+}\, \widehat{o(f)}\big] = 1\ \ lubnd(S, a, s, v) = l \le a \le a + \sigma(s) \times \langle E_v[v] \rangle_U - 1 \le u$$

$$\begin{aligned} alloc(S, a, p, s, v) = \Big(&B\Big[\big\{\widehat{a+i} \leftarrow x_i \big| 0 \le i < \sigma(s) \times \langle E_v[v] \rangle_U, x_i \in \mathbb{B}^d\big\}\Big], \\ &V\Big[\big\{\widehat{a+i} \leftarrow 1 \big| 0 \le i < \sigma(s) \times \langle E_v[v] \rangle_U\big\}\Big], \\ &L\big[\widehat{a} \leftarrow \widehat{\sigma(s)} \,\widehat{\times}\, E_v[v]\big], E_p[p \leftarrow \widehat{a}], E_v\Big) \end{aligned}$$

$$free(S, p) = \Big(B, V\Big[\big\{E_p[p] \,\widehat{+}\, \widehat{i} \leftarrow 0 \big| 0 \le i < \langle L\big[E_p[p]\big] \rangle_U\big\}\Big], L\big[E_p[p] \leftarrow \widehat{0}\big], E_p, E_v\Big)$$

$$fresh(S, a, s, v) = \forall i \in \mathbb{N}\mathbf{.}\ i < \sigma(s) \times \langle E_v[v] \rangle_U \implies V[\widehat{a+i}] = 0$$

$$allocated(S, p, l) = \forall i \in \mathbb{N}\mathbf{.}\ i < l \implies V\big[E_p[p] \,\widehat{+}\, \widehat{i}\big] = 1$$

$$mayalloc(S, a, s, v) = lubnd(S, a, s, v) \wedge fresh(S, a, s, v) \wedge L[\widehat{a}] = \widehat{0}$$

$$mayfree(S, p) = L\big[E_p[p]\big] \ne \widehat{0} \wedge allocated(S, p, \langle L\big[E_p[p]\big] \rangle_U)$$

Рис. 4.2. Обозначения, используемые в исходной семантике базового языка

(включая её саму), адресуемую указателем p и имеющею адрес начала a. С помощью функции π можно определить отношение $\sqsubset^+$ транзитивной вложенности структур. Тогда условие корректного определения размера структуры s имеет вид $s \not\sqsubset^+ s$.

Для целочисленных и указательных полей размер полагается равным 1, для полей-массивов структур — произведению размера соответствующей структуры на число элементов в массиве, размер всей структуры полагается равным сумме размеров всех её полей. Размер поля или структуры — всегда натуральное число (включая 0). Для полей структур введем также отображение $o : \mathbb{F} \to \mathbb{N}$, задающее смещение поля относительно начала структуры. Смещение поля $f \in \varphi(s)$ полагается равным сумме размеров полей в наибольшем префиксе структуры s, в который еще не входит поле f.

Для рассмотрения операций над представлениями чисел и указателей в виде битовых векторов длины d введем обозначения $\langle x \rangle_U$ и $\langle x \rangle_S$ для беззнакового (натурального) и знакового (целого) числового значения битового вектора $x \in \mathbb{B}^d$ в предположении, что используется представление по модулю 2^d в дополнительном коде. Соответствующий целому числу x по модулю 2^d битовый вектор будем обозначать символом $\widehat{x}$. Введем также обозначения $\widehat{+}, \widehat{-}$ и $\widehat{\times}$ для соответствующих операций в дополнительном коде, $\widehat{\div}_S$ для операции знакового целочисленного деления и $\widehat{\lozenge}$ для знаковых отношений сравнения соответствующих битовых представлений.

Множество значений для конечных последовательностей инструкций базового языка состоит из двух элементов — $\top$ и $\bot$. Значение $\top$ соответствует корректности соответствующей последовательности инструкций относительно всех содержащихся в ней проверочных утверждений о значениях переменных (инструкций $\mathbf{assert}(v \lozenge 0)$), а также свойств безопасности доступа к памяти, а именно разыменования только выделенной памяти, корректности всех операций освобождения и отсутствия утечек памяти. Корректность проверяется в предположениях, задаваемых инструкциями предположения $\mathbf{assume}(v \lozenge 0)$, в случае невыполненности одного из которых последовательность инструкций считается корректной, а результат вычисления — равным $\top$. Результат $\top$ также соответствует нехватке памяти для продолжения вы-

$$\text{arith}\ \ v = i;\, oo\big|_S \blacktriangleright oo\big|_{upd^v}(S, v, \widehat{[\![i]\!]_{E_v}}) \qquad \text{null}\ \ p = \mathrm{NULL};\, oo\big|_S \blacktriangleright oo\big|_{upd^p}(S, p, \widehat{0}) \qquad \frac{\text{top}\ \ init(S)}{\varepsilon\big|_S \blacktriangleright \top}$$

$$\frac{\text{diff}\ \ p_1, p_2 : \texttt{pointer}\, s\, \rho}{v = p_1 - p_2;\, oo\big|_S \blacktriangleright oo\big|_{upd^v}\big(S, v, (E_p[p_1] \mathbin{\widehat{-}} E_p[p_2]) \mathbin{\widehat{\div}_S} \widehat{\sigma(s)}\big)} \qquad \frac{\text{bot}\ \ \neg init(S)}{\varepsilon\big|_S \blacktriangleright \bot}$$

$$\text{eq}\ \ v = p_1 == p_2;\, oo\big|_S \blacktriangleright oo\big|_{upd^v}\big(S, v, \mathrm{ite}(E_p[p_1] = E_p[p_2], \widehat{1}, \widehat{0})\big) \qquad \text{toint}\ \ \mathbf{to\_int}(p, v);\, oo\big|_S \blacktriangleright oo\big|_S$$

$$\frac{\text{shift}\ \ p_2 : \texttt{pointer}\, s\, \rho}{p_1 = p_2 + v;\, oo\big|_S \blacktriangleright oo\big|_{upd^p}\big(S, p_1, E_p[p_2] \mathbin{\widehat{+}} E_v[v] \mathbin{\widehat{\times}} \widehat{\sigma(s)}\big)} \qquad \text{ofint}\ \ \mathbf{of\_int}(p, s, v);\, oo\big|_S \blacktriangleright oo\big|_S$$

$$\text{field}\ \ p_1 = \&p_2\texttt{->}f_s;\, oo\big|_S \blacktriangleright oo\big|_{upd^p}\big(S, p_1, E_p[p_2] \mathbin{\widehat{+}} \widehat{o(f_s)}\big) \qquad \text{mayalias}\ \ \mathbf{may\_alias}(p_1, p_2);\, oo\big|_S \blacktriangleright oo\big|_S$$

$$\text{container}\ \ p_1 = \mathbf{container\_of}(p_2, f_{s'});\, oo\big|_S \blacktriangleright oo\big|_{upd^p}\big(S, p_1, E_p[p_2] \mathbin{\widehat{-}} \widehat{o(f_{s'})}\big)$$

$$\frac{\text{deref}\ \ x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad valid(S, p_1, f_x)}{x = p_1\texttt{->}f_x;\, oo\big|_S \blacktriangleright oo\big|_{upd^x}\big(S, x, B\big[E_p[p_1] \mathbin{\widehat{+}} \widehat{o(f_x)}\big]\big)} \qquad \frac{\text{deref}^{\bot}\ \ x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad \neg valid(S, p_1, f_x)}{x = p_1\texttt{->}f_x;\, oo\big|_S \blacktriangleright \bot}$$

$$\frac{\text{assign}\ \ x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad valid(S, p_1, f_x)}{p_1\texttt{->}f_x = x;\, oo\big|_S \blacktriangleright oo\big|_{upd^B}(S, E_p[p_1] \mathbin{\widehat{+}} \widehat{o(f_x)}, E_x[x])} \qquad \frac{\text{assign}^{\bot}\ \ x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad \neg valid(S, p_1, f_x)}{p_1\texttt{->}f_x = x;\, oo\big|_S \blacktriangleright \bot}$$

$$\frac{\text{alloc}\ \ p : \texttt{pointer}\, s\, \rho \qquad \exists a \in \mathbb{N}\texttt{.}\ mayalloc(S, a, s, v)}{p = \mathbf{alloc}(v);\, oo\big|_S \blacktriangleright oo\big|_{alloc(S, a, p, s, v)}} \qquad \frac{\text{alloc}^{\top}\ \ p : \texttt{pointer}\, s\, \rho \qquad \forall a \in \mathbb{N}\texttt{.}\ \neg mayalloc(S, a, s, v)}{p = \mathbf{alloc}(v);\, oo\big|_S \blacktriangleright \top}$$

$$\frac{\text{free}\ \ mayfree(S, p)}{p = \mathbf{free}(p);\, oo\big|_S \blacktriangleright oo\big|_{free(S, p)}} \qquad \frac{\text{free}^{\bot}\ \ \neg mayfree(S, p)}{p = \mathbf{free}(p);\, oo\big|_S \blacktriangleright \bot}$$

$$\frac{\text{assert}\ \ E_v[v] \mathbin{\widehat{\Diamond}} \widehat{0}}{\mathbf{assert}(v \Diamond 0);\, oo\big|_S \blacktriangleright oo\big|_S} \qquad \frac{\text{assert}^{\bot}\ \ \neg E_v[v] \mathbin{\widehat{\Diamond}} \widehat{0}}{\mathbf{assert}(v \Diamond 0);\, oo\big|_S \blacktriangleright \bot}$$

$$\frac{\text{assume}\ \ E_v[v] \mathbin{\widehat{\Diamond}} \widehat{0}}{\mathbf{assume}(v \Diamond 0);\, oo\big|_S \blacktriangleright oo\big|_S} \qquad \frac{\text{assume}^{\top}\ \ \neg E_v[v] \mathbin{\widehat{\Diamond}} \widehat{0}}{\mathbf{assume}(v \Diamond 0);\, oo\big|_S \blacktriangleright \top}$$

Рис. 4.3. Исходная семантика базового языка

числений. Верификация в предположении о достаточном объеме доступной памяти позволяет избежать необходимости формального доказательства корректности некоторого верхнего приближения совокупного объема используемой памяти для каждой верифицируемой программы (что в предлагаемой модели памяти существенно осложнило бы верификацию). Однако в таком предположении последовательности инструкций (соответствующие путям выполнения программ), выделяющие объем памяти, превышающий размеры доступного адресного пространства, становятся неотличимыми с точки зрения результата вычисления от корректных последовательностей. Это является одним из ограничений предлагаемой модели памяти. При необходимости для снятия этого ограничения модель может быть расширена подсчетом суммарного объема выделенной памяти. На практике в инструменте верификации на базе инструкции $\mathbf{alloc}(v)$ моделируется недетерминированная функция выделения памяти (которая может недетерминированно возвращать результат NULL), а для значения аргумента v вводится ограничение, позволяющее включить в рассмотрение все существенные пути выполнения (то есть предотвратить отсечение реальных ошибочных базовых (ациклических) путей выполнения за счет выделения на них достаточно большого объема памяти). Значение $\bot$ соответствует нарушению одного из проверочных условий либо одного из свойств безопасности доступа к памяти. Состояния вычисления для последовательностей инструкций принадлежат объединению множества значений $\{\top, \bot\}$ и множества промежуточных состояний вида $oo\big|S$, где $S = (B, V, L, E_p, E_v)$ — состояние памяти, включающее $E_v : \mathbb{V} \to \mathbb{B}^d$ — память для целочисленных переменных, $E_p : \mathbb{P} \to \mathbb{B}^d$ — память для указательных переменных, $B : \mathbb{B}^d \to \mathbb{B}^d$ — кучу (память, адресуемую указателями), $V : \mathbb{B}^d \to \mathbb{B}$ — карту выделенности адресов и $L : \mathbb{B}^d \to \mathbb{B}^d$ — карту размеров выделенных блоков. Отношение вычисления $\blacktriangleright$ для исходной семантики (соответствует одному шагу вычисления в мелкошаговой (англ. *small-step*) опера-

ционной семантике [101, 77]) задается правилами вывода, представленными на рис. 4.3. На рисунке используются следующие обозначения (соответствующие определения приведены на рис. 4.2): $[\![i]\!]_{E_v}$ — значение арифметического выражения i над константами и переменными, вычисленное с использованием значений целочисленных переменных из памяти E_v; $upd^v(S, v, i)$ — состояние памяти, соответствующее результату обновления значения переменной v в памяти E_v (на рис. 4.2 используется обозначение обновления отображения $E[v \leftarrow i]$, означающее отображение E', отличное от E только для аргумента v, который E' отображает в значение i); $upd^p(S, p, a)$ — аналогичный результат обновления значения указателя p в памяти E_p; $upd^B(S, a, v)$ — обновление значения ячейки памяти B по адресу a; $\mathrm{ite}(c, t, f)$ обозначает t, если предикат c выполнен и f в противном случае; $valid(S, p, f)$ — условие выделенности ячейки памяти, в которой располагается поле $f \in \mathbb{F}_v \cup \mathbb{F}_p$ структуры, адресуемой указателем p; $mayalloc(S, a, s, v)$ — условие наличия свободной памяти в диапазоне между адресами l и u включительно для выделения массива структур с тегом s размера v по адресу a; $alloc(S, a, p, s, v)$ — результат выделения памяти аналогично $mayalloc$ с записью адреса a в указательную переменную p (обозначение $B\big[\{a(\bar{x}) \leftarrow v(\bar{x}) \big| P(\bar{x})\}\big]$ используется для множественного обновления отображения B, при котором в новом отображении каждому значению $a(\bar{x})$ для любых значений переменных из набора $\bar{x}$, удовлетворяющих $P(\bar{x})$, ставится в соответствие значение $v(\bar{x})$; значения x_i, записываемые в выделенную память, выбираются недетерминированно); $mayfree(S, p)$ — условие корректности для операции освобождения памяти, адресуемой указателем p (в целях упрощения случай $p == \mathrm{NULL}$ считается некорректным); $free(S, p)$ — результат освобождения памяти, адресуемой указателем p; $init(S)$ — предикат, задающий начальные предположения для значений карт выделенности адресов V и размеров блоков L.

Правила вывода, представленные на рис. 4.3, обеспечивают возможность

однозначно с точностью до выбора недетерминированных значений x_i и адресов в правиле alloc выполнить шаг вычисления из любого состояния, так как всем правилам вывода, имеющим предусловия, отличные от тождественной истины, соответствуют дополняющие их правила вывода с индексом $\bot$ или $\top$, предусловия которых являются отрицаниями соответствующих основных предусловий, и каждой инструкции базового языка соответствует либо одно правило вывода с тождественно истинным предусловием (для корректно типизированной последовательности инструкций), либо пара дополняющих друг друга правил вывода. Кроме этого, в результате применения каждого правила вывода, за исключением top и bot, длина последовательности инструкций уменьшается на 1. Обозначим через $\blacktriangleright^*$ наименьшее рефлексивное транзитивное отношение, содержащее $\blacktriangleright$. Тогда по перечисленным причинам отношение $\blacktriangleright^*$ однозначно связывает любое начальное состояние вычисления с одним или двумя (в силу недетерминизма) элементами множества значений $\{\top, \bot\}$. Таким образом, верна следующая теорема:

Теорема 1. *Для любой конечной последовательности инструкций oo базового языка и любого начального состояния вычисления* $oo\big|S$ *отношение вычисления* $\blacktriangleright^*$ *однозначно определяет соответствующее непустое подмножество всех возможных значений из множества* $\{\top, \bot\}$ *(то есть одно из множеств* $\{\top\}$, $\{\bot\}$ *или* $\{\top, \bot\}$*).*

Будем далее использовать обозначение $oo \blacktriangleright^* \top$, эквивалентное утверждению о том, что для любого S выполнено $\forall r \in \{\top, \bot\}\,.\; init(S) \wedge oo\big|S \blacktriangleright^* r \implies r = \top$, а также обозначение $oo \blacktriangleright^* \bot$, эквивалентное утверждению о существовании некоторого S, для которого выполнено $init(S) \wedge oo\big|S \blacktriangleright^* \bot$.

4.2. Нормализация Си-программ

Рассмотрим теперь преобразования, позволяющие свести произвольные линейные фрагменты Си-кода, использующие объединения, префиксные и непрефиксные приведения типов указателей на структуры, а также конструкции получения указателя на объемлющую структуру, к соответствующим последовательностям инструкций базового языка. Так как рассматриваемый базовый язык предназначен для использования в инструменте дедуктивной верификации (отсюда неизбежное наличие задаваемых пользователем аннотаций), допустимо предположение, что исходная Си-программа аннотирована двумя видами специально вводимых спецификаций — `//@ reinterpret_union(`p_u`,` f_1`,` f_2`)` и `//@ reinterpret_struct(`$p + (0..n)$`,` s_1`,` s_2`)`. Не ограничивая общности рассуждений, будем считать, что все указатели в программе являются указателями на структуры или объединения и адресуют кучу, а также что все поля объединений являются структурами (или массивами структур). Соответствующие нормализующие преобразования Си-программ описаны в статьях [105, 106, 5]. Кроме этого, в целях упрощения дальнейшего рассмотрения будем предполагать, что все целочисленные типы в программах имеют размер, совпадающий с размером указателя и в структурах отсутствуют выравнивания (вместо выравниваний можно использовать явные целочисленные поля, а расширение базового языка поддержкой целочисленных типов различных размеров усложнит его описание, но несущественно не повлияет на моделирование операций с указателями). В указанных предположениях возможны преобразования, перечисленные далее.

Для получения указателя на объемлющую структуру можно использовать соответствующую конструкцию базового языка. Например, для фрагмента Си-кода вида

```
(struct s *) ((char *) p - offsetof(struct s, f_s.field))
```

в контексте, где $\mathbb{S} \supseteq \{s, s'\}$, $\mathbb{F}_s \supseteq \{f_{s.field}\}$, $\varphi(s) \supseteq \{f_{s.field}\}$, $\mathbb{P} \supseteq \{p, p'\}$, $f_{s.field} : \texttt{struct}\, s'\, [c]$, $p : \texttt{pointer}\, s'\, \rho$, $p' : \texttt{pointer}\, s\, \rho'$, соответствующая конструкция базового языка:

$$p' = \textbf{container\_of}(p, f_{s.field}) \ .$$

Для произвольного объединения фрагмент вида

```
union u {
  struct s_1 {
    t_{1,1} f_{1,1};
    // ...
    t_{1,n_1} f_{1,n_1};
  } s_1;
  // ...
  struct s_m {
    t_{m,1} f_{m,1};
    // ...
    t_{m,n_m} f_{m,n_m};
  } s_m;
};
// ...
union u *p_u;
/*...*/ /*(1)*/ &p_u->s_i /*...*/
/*...*/ /*(2)*/ (struct s_i *) p_u /*...*/
/*...*/ /*(3)*/ p_u->s_i.f_{i,j} /*...*/
/*...*/ /*(4)*/ p_u->s_i.f_{l,k} /*...*/
// ...
struct s_i *p_s;
/*...*/ /*(5)*/ (union u *) p_s /*...*/
```

в контексте, где $\mathbb{S} \supseteq \{\text{void}, s_1, ..., s_m\}$, $\mathbb{F}_p \cup \mathbb{F}_v \cup \mathbb{F}_s \supseteq \{f_{1,1}, \ldots, f_{m,n_m}, f_{s_1.\text{void}}, \ldots, f_{s_m.\text{void}}\}$, $f_{i,j} \in \mathbb{F}_v \cup \mathbb{F}_p$, $f_{l,k} \in \mathbb{F}_s$ ($f_{l,k}$ — полемассив структур), $\mathbb{P} \supseteq \{p_u, p_s, p', p'', p'''\}$, $\varphi(s_i) = \{f_{s_i.\text{void}}, f_{i,1}, \ldots, f_{i,n_i}\}$, $1 \leq i \leq m$, $p_u, p''' : \texttt{pointer}\, \text{void}\, \rho$, $p', p_s : \texttt{pointer}\, s_i\, \rho'$, $x \in \mathbb{P} \cup \mathbb{V}$, последовательности инструкций базового языка, соответствующие конструкциям

(1) – (5):

$$\begin{array}{ll}(1) & p' = \textbf{container\_of}(p_u, f_{s_i.\text{void}}) \\ (2) & p' = \textbf{container\_of}(p_u, f_{s_i.\text{void}}) \\ (3) & p' = \textbf{container\_of}(p_u, f_{s_i.\text{void}}); \\ & x = p'\texttt{->}f_{i,j} \\ (4) & p' = \textbf{container\_of}(p_u, f_{s_i.\text{void}}); \\ & p'' = \texttt{\&}p'\texttt{->}f_{l,k} \\ (5) & p''' = \texttt{\&}p_s\texttt{->}f_{s_i.\text{void}}.\end{array}$$

То есть указатель на объединение заменяется указателем на структуру с тегом void без полей, взятие адреса структуры-поля объединения и приведение типа указателя на объединение к указателю на структуру-поле этого объединения (в общем случае — произвольную структуру) выражается с помощью конструкции получения указателя на объемлющую структуру (все структуры, кроме void содержат первое поле $f_{s.\text{void}}$), а приведение типа указателя на структуру-поле объединения (вообще говоря, на любую структуру) к типу указателя на объединение выражается с помощью конструкции получения указателя на вложенный массив структур — поле $f_{s.\text{void}}$: `struct` void `[1]` ($\sigma(f_{s.\text{void}}) = 0$, т.к. размер структуры void без полей равен 0).

В соответствии с исходной семантикой обращения к полям структур объединения, представленного на базовом языке рассмотренным способом, возможны в любом состоянии вычисления, где выделена адресуемая соответствующими указателями память. В модельной семантике, представленной далее в разделе 4.3, каждая ячейка памяти, в частности, памяти объединения, в каждом состоянии вычисления может быть доступна только через указатель на какой-либо один из типов структур. Для переключения между интерпретациями ячеек памяти как полей различных структур в базовый язык введены инструкции $\textbf{to\_int}(p, v)$ и $\textbf{of\_int}(p, s, v)$, которые в исходной семантике интерпретируются как пустые инструкции. Для чтения и записи в поля структур объединения предполагается использование в исходном аннотированном Си-коде специальной спецификационной конструкции `//@ reinterpret_union(`p_u`,` f_1`,` f_2`)`, которая позволяет переключаться

между структурами объединения так, что в каждом состоянии программы чтение либо запись возможны в поля одной и только одной из структур объединения. Например, для фрагмента вида

```
/*...*/ /*(1)*/ p_u->s_i.f_{i,j} /*...*/
/*...*/ /*(2)*/ p_u->s_i.f_{i,j} = e_1 /*...*/
/*(3)*/ //@ reinterpret_union(p_u, s_i, s_k);
/*...*/ /*(4)*/ p_u->s_k.f_{k,l} /*...*/
/*...*/ /*(5)*/ p_u->s_k.f_{k,l} = e_2 /*...*/
```

соответствующая последовательность инструкций базового языка (в соответствующем контексте):

$$
\begin{array}{ll}
(1) & p' = \textbf{container\_of}(p_u, f_{s_i.\mathrm{void}}); \\
 & x = p'\text{->}f_{i,j} \\
(2) & p' = \textbf{container\_of}(p_u, f_{s_i.\mathrm{void}}); \\
 & p'\text{->}f_{i,j} = x_{e_1} \\
(3) & p' = \textbf{container\_of}(p_u, f_{s_i.\mathrm{void}}); \\
 & v = 1; \\
 & \textbf{to\_int}(p', v); \\
 & p'' = \textbf{container\_of}(p_u, f_{\mathrm{int.void}}); \\
 & \textbf{of\_int}(p'', s_k, v) \\
(4) & p''' = \textbf{container\_of}(p_u, f_{s_k.\mathrm{void}}); \\
 & x' = p'''\text{->}f_{k,l} \\
(5) & p''' = \textbf{container\_of}(p_u, f_{s_k.\mathrm{void}}); \\
 & p'''\text{->}f_{l,k} = x_{e_2}.
\end{array}
$$

Таким образом, переключение между структурами объединения в базовом языке производиться с помощью конструкций $\textbf{to\_int}(p, v)$ и $\textbf{of\_int}(p, s, v)$ через промежуточный массив структур с тегом int и одним полем $f_{\mathrm{int.int}}$.

Для произвольного префиксного приведения типа указателя на структуру

```
struct s_o {
  struct s_i {
    t_{i,1} f_{i,1};
    // ...
    t_{i,n_i} f_{i,n_i};
  } s_i;
  t_{o,1} f_{o,1};
  // ...
  t_{o,n_o} f_{o,n_o};
};
```

```
// ...
struct s_o *p_o;
/*...*/ /*(1)*/ (struct s_i *) p_o /*...*/
// ...
struct s_i *p_i;
/*...*/ /*(2)*/ (struct s_o *) p_i /*...*/
```

соответствующие восходящее приведение типа можно рассматривать как получение указателя на вложенную структуру, а нисходящее — как получение указателя на объемлющую:

$$(1)\quad p_i = \&p_o\texttt{->}s_i$$
$$(2)\quad p_o = \textbf{container\_of}(p_i, f_{s_o.s_i}).$$

Произвольные непрефиксные приведения типа указателя на структуру

```
struct s_1 {
  t_{1,1} f_{1,1};
  // ...
  t_{1,n_1} f_{1,n_1};
} *p_1;
struct s_2 {
  t_{2,1} f_{2,1};
  // ...
  t_{2,n_2} f_{2,n_2};
};
/*...*/ /*(1)*/ p_1->f_{1,i} /*...*/
/*...*/ /*(2)*/ p_1->f_{1,i} = e_1 /*...*/
/*(3)*/ //@ reinterpret_struct(p_1 + (0..k), s_1, s_2);
/*...*/ /*(4)*/ ((struct s_2 *) p_1)->f_{2,j}
/*...*/ /*(5)*/ ((struct s_2 *) p_1)->f_{2,j} = e_2 /*...*/
```

можно рассматривать аналогично использованию объединений соответствующих структур. Основное отличие заключается в том, что при непрефиксном приведении типа размеры структур могут различаться, в том числе быть не кратными друг другу и, таким образом, в общем виде при выполнении операции `//@ reinterpret_struct(`p_1`+(0..`k`), `s_1`, `s_2`)` для подмассива структур возможно переключение разрешения доступа к остаточному конечному отрезку множества адресов $p_1 + (0..k)$ на структуру с тегом int. В этой секции рассмотрим частный случай кратного размера, наиболее общий случай рассмотрен в секции 4.3 в доказательстве теоремы о полноте модельной семантики. Для

случая кратного размера соответствующая последовательность инструкций:

$$
\begin{array}{ll}
(1) & x = p_1\texttt{->}f_{1,j} \\
(2) & p_1\texttt{->}f_{1,j} = x_{e_1} \\
(3) & v = k; \\
 & \textbf{to\_int}(p_1, v); \\
 & p = \&p_1\texttt{->}f_{s_1.\mathrm{void}}; \\
 & p' = \textbf{container\_of}(p, f_{\mathrm{int.void}}); \\
 & v = k \times \sigma(s_1)/\sigma(s_2); \\
 & \textbf{of\_int}(p', s_2, v) \\
(4) & p = \&p_1\texttt{->}f_{s_1.\mathrm{void}}; \\
 & p_2 = \textbf{container\_of}(p, f_{s_2.\mathrm{void}}); \\
 & x'' = p_2\texttt{->}f_{2,j} \\
(5) & p = \&p_1\texttt{->}f_{s_1.\mathrm{void}}; \\
 & p_2 = \textbf{container\_of}(p, f_{s_2.\mathrm{void}}); \\
 & \&p_2\texttt{->}f_{2,j} = x_{e_2}.
\end{array}
$$

Поля-объединения и поля-массивы объединений в структурах можно заменить полями-структурами или массивами структур, выбрав в каждом объединении одно из полей-структур наибольшего размера. При этом обращения к остальным (не выбранным) полям-структурам вложенных объединений можно преобразовывать в инструкции базового языка аналогично обращениям к полям-структурам объединений верхнего уровня.

4.3. Модельная семантика базового языка

Модельная семантика базового языка описывает результат трансляции программы на базовом языке в модель, для которой генерируются условия верификации. Основная цель введения модельной семантики отличной от исходной — эффективное кодирование операций с указателями, позволяющее избежать необходимости использования в условиях верификации большого числа явных условий непересечения адресов и большого числа неявно генерируемых решателями экземпляров утверждений о сохранении предыдущих значений при обновлении массивов, представляющих состояние областей памяти программы. Для достижения более эффективного кодирования в модельной се-

мантике, в отличие от исходной, адресуемая указателями память программы (куча) B разделяется на совокупность логических массивов [5, 143] M. В совокупности массивов M также хранятся и значения, соответствующие исходным отображениям V и L. Для представления устройства совокупности массивов M введем дополнительные обозначения и ограничения, приведенные на рис. 4.4. Для хранения в M значения исходных отображений V и L будем рассматривать расширенное множество полей $\mathbb{F}_{vp}$, в которое помимо всех целочисленных и указательных полей из $\mathbb{F}_v \cup \mathbb{F}_p$ входят поля вида $f_{s.\text{valid}}$ и $f_{s.\text{len}}$ типа `int` для каждой структуры, кроме void. Для удобного введения разделения памяти программы одновременно по типам структур и регионам будем считать, что каждому региону ρ из множества $\mathbb{R}$ соответствует единственный тег структуры $\tau(\rho)$. Это вводит дополнительное ограничение на контекст типизации термов базового языка Γ, так как для корректного задания функции τ термам-указателям на структуры с различными тегами контекстом Γ должны быть сопоставлены различные регионы. Введем также вспомогательное обозначение $\Gamma_\rho(p)$ для регионов указательных термов из $\mathcal{T}^*$ так, что для любого такого терма p выполнено $\Gamma(p) = \texttt{pointer}\, s\, \big(\Gamma_\rho(p)\big)$ для некоторого тега s. Совокупность массивов M представляет собой отображение типа $\mathbb{R} \times \mathbb{B}^d \times \mathbb{F}_{vp} \to \mathbb{B}^d$, то есть отображение троек вида (регион, адрес, поле) в битовые представления соответствующих значений. В силу конечности множества $\mathbb{F}_{vp}$, при условии конечности множества $\mathbb{R}$ (обеспечивается алгоритмом вывода регионов, см. секцию 4.4) совокупность массивов M состоит из конечного числа логических массивов, которые используются для моделирования состояния памяти программы на базовом языке. Каждый логический массив соответствует паре вида (f, ρ), где $f \in \varphi(\tau(\rho)) \cup \{f_{s.\text{valid}}, f_{s.\text{len}}\}$, $\tau(\rho) \neq \text{void}$. Отображения-памяти для целочисленных и указательных переменных в модельной семантике полностью аналогичны соответствующим отображениями в исходной семантике. Состояние вычисления последователь-

ности инструкций базового языка в модельной семантике принадлежит множеству $\{\top, \bot\} \cup \left\{ oo|S' \middle| S' = (M, E_p, E_v) \right\}$, где значения $\top$ и $\bot$ совпадают с соответствующими значениями в исходной семантике.

$$\mathbb{F}_{vp} = \mathbb{F}_v \cup \mathbb{F}_p \cup \{f_{s.\text{valid}}, f_{s.\text{len}} | s \in \mathbb{S}, s \neq \text{void}\}, \quad f_{s.\text{len}}, f_{s.\text{valid}} : \texttt{int},$$

$$\forall p_1, p_2 \in \mathcal{T}^*, s_1, s_2 \in \mathbb{S}, \rho_1, \rho_2 \in \mathbb{R}.$$

$$\Gamma(p_1) = \texttt{pointer}\, s_1\, \rho_1 \wedge \Gamma(p_2) = \texttt{pointer}\, s_2\, \rho_2 \wedge s_1 \neq s_2 \Rightarrow \rho_1 \neq \rho_2,$$

$$\Gamma_\rho : \mathbb{T} \to \mathbb{R}, \quad \tau : \mathbb{R} \to \mathbb{S}, \quad \forall p \in \mathcal{T}^*.\, p : \texttt{pointer}\, s\, \rho \iff \Gamma_\rho(p) = \rho \wedge \tau(\rho) = s,$$

$$M : \mathbb{R} \times \mathbb{B}^d \times \mathbb{F}_{vp} \to \mathbb{B}^d, \quad E_p : \mathbb{P} \to \mathbb{B}^d, \quad E_v : \mathbb{V} \to \mathbb{B}^d, \quad S' = (M, E_p, E_v),$$

$$init'(S') = \forall \rho \in \mathbb{R}, \underline{a \in \mathbb{B}^d}.\, M[(\rho, a, f_{\tau(\rho).\text{valid}})] = \widehat{0} \wedge M[(\rho, a, f_{\tau(\rho).\text{len}})] = \widehat{0}$$

$$valid'(S', p, s) = M[(\Gamma_\rho(p), E_p[p], f_{s.\text{valid}})] = \widehat{1}$$

$$lubnd(S', a, s, v) = l \leq a \leq a + \sigma(s) \times \langle E_v[v] \rangle_U - 1 \leq u \quad upd'^v(S', v, i) = (M, E_p, E_v[v \leftarrow i])$$

$$upd'^p(S', p, a) = (M, E_p[p \leftarrow a], E_v) \quad upd'^M(S', p, f, v) = \left(M[(\Gamma_\rho(p), E_p[p], f) \leftarrow v], E_p, E_v\right)$$

$$valid^*(S', a, p, s, l, v) = \forall \underline{i \in \mathbb{N}}.\, i < l \Rightarrow$$

$$\forall s' \in \mathbb{S}, p' \in \mathbb{T}, a' \in \mathbb{Z}.\, (s', p', a') \in \pi\left(s, p, a + i \times \sigma(s)\right) \Rightarrow M[(\Gamma_\rho(p'), \widehat{a'}, f_{s'.\text{valid}})] = v$$

$$fresh'(S', a, p, s, l) =$$

$$E_v[v] \neq \widehat{0} \Rightarrow \forall \rho \in \mathbb{R}, \underline{i \in \mathbb{Z}}.\, -\sigma\left(\tau(\rho)\right) < i < l \times \sigma(s) \Rightarrow M[(\rho, \widehat{a + i}, f_{\tau(\rho).\text{valid}})] = \widehat{0}$$

$$mayalloc'(S', a, p, s, v) =$$

$$lubnd(S', a, s, v) \wedge fresh'(S', a, p, s, \langle E_v[v] \rangle_U) \wedge M[(\Gamma_\rho(p), \widehat{a}, f_{s.\text{len}})] = \widehat{0}$$

$$mayfree'(S', p, s) =$$

$$M[(\Gamma_\rho(p), E_p[p], f_{s.\text{len}})] \neq \widehat{0} \wedge valid^*(S', E_p[p], p, s, \langle M[(\Gamma_\rho(p), E_p[p], f_{s.\text{len}})] \rangle_U, \widehat{1})$$

$$upd'^M_{f_{valid}}(S', a, p, s, l, v) = \left\{ \left(\Gamma_\rho(p'), \widehat{a'}, f_{s'.\text{valid}}\right) \leftarrow v \middle| \underline{0 \leq i < l}, (s', p', a') \in \pi\left(s, p, a + i \times \sigma(s)\right) \right\}$$

$$fields(a, p, s) = \bigcup_{f \in \varphi(s)} \begin{cases} \bigcup_{i=0}^{c-1} fields\left(a + o(f) + i \times \sigma(s'), \texttt{\&p->}f, s'\right), f : \texttt{struct}\, s'\, [c] \in \mathbb{F}_s, \\ (p, a, a + o(f), f), f \in \mathbb{F}_v \cup \mathbb{F}_p \end{cases}$$

$$alloc'(S', a, p, s, v) = \Big(M\Big[upd'^M_{f_{valid}}(S', a, p, s, \langle E_v[v] \rangle_U, \widehat{1}) \cup \left\{(\Gamma_\rho(p), \widehat{a}, f_{s.\text{len}}) \leftarrow E_v[v]\right\} \cup$$

$$\left\{(\Gamma_\rho(p'), \widehat{a'}, f_x) \leftarrow x_{i,f_x} \middle|\right.$$

$$\left. \underline{0 \leq i < \langle E_v[v] \rangle_U}, (p', a', a'', f_x) \in fields(a + i \times \sigma(s), p, s), x_{i,f_x} \in \mathbb{B}^d \right\}\Big],$$

$$E_p[p \leftarrow \widehat{a}], E_v\Big)$$

$$free'(S', p, s) = \Big(M\Big[upd'^M_{f_{valid}}(S', E_p[p], p, s, \langle M[(\Gamma_\rho(p), E_p[p], f_{s.\text{len}})] \rangle_U, \widehat{0}) \cup$$

$$\left\{(\Gamma_\rho(p), E_p[p], f_{s.\text{len}}) \leftarrow \widehat{0}\right\}\Big], E_p, E_v\Big)$$

Рис. 4.4. Обозначения, используемые в модельной семантике базового языка (начало)

Отношение вычисления $\triangleright$ для модельной семантики задается правилами вывода, представленными на рис. 4.6. Приведенным правилам вывода может быть непосредственно сопоставлена соответствующая аксиоматическая семантика в виде пред- и постусловий соответствующих абстрактных операций, для которых в свою очередь инструментом дедуктивной верификации могут быть сгенерированы соответствующие условия верификации. При этом каждая абстрактная операция соответствует либо одному правилу вывода с тождественно истинным предусловием, либо паре правил вывода с дополняющими предусловиями (являющимися логическими отрицаниями друг друга), одно из которых имеет индекс $\top$ или $\bot$. Если правило вывода с дополняющим предусловием имеет индекс $\bot$, то предусловие соответствующего правила без индекса совпадает с предусловием соответствующей абстрактной операции. В случае индекса $\top$ предусловие правила без индекса совпадает с одним из постусловий соответствующей операции. Любая абстрактная операция (кроме соответствующей паре правил top и bot) также имеет постусловие, специфицирующее результат обновления состояния памяти в результате её успешного выполнения, который совпадает с результатом обновления памяти из состояния вычисления справа от отношения $\triangleright$ в следствии соответствующего правила вывода. В формулах, приведенных на рис. 4.4 и 4.5 подчеркнуты кванторы, присутствующие в результирующих контрактах соответствующих абстрактных операций после раскрытия кванторов по конечным множествам. Не выделенные подчеркиванием кванторы могут быть статически раскрыты инструментом верификации во время трансляции. Большинство правил вывода для отношения $\triangleright$ получаются непосредственно из соответствующих правил вывода для отношения $\blacktriangleright$ (рис. 4.3) подстановкой модельного состояния памяти S' вместо исходного состояния S и обозначений $init'(S')$, $upd'^{v}(S', v, i)$ и $upd'^{p}(S', p, a)$ вместо $init(S)$, $upd^{v}(S, v, i)$ и $upd^{p}(S, p, a)$ соответственно (такие правила вывода перечислены на рис. 4.6). Рассмотрим обозначения, ис-

пользуемые в модельной семантике базового языка (рис. 4.6), приведенные на рис. 4.4 и 4.5.

$$cast(p, s_1, s_2) = \mathbf{container\_of}(\&p\text{->}f_{s_1.\mathrm{void}}, f_{s_2.\mathrm{void}}) \quad fct(s, s', l) = \begin{cases} l, s \neq \mathrm{int}, \\ l \times \sigma(s'), s = \mathrm{int} \end{cases}$$

$$upd2(S', a, p_1, p_2, s_1, s_2, l) =$$
$$upd'^{M}_{f_{valid}}(S', a, p_1, s_1, fct(s_1, s_2, l), \widehat{0}) \cup upd'^{M}_{f_{valid}}(S', a, p_2, s_2, fct(s_2, s_1, l), \widehat{1})$$

$$maytoint(S', p, s, v) =$$
$$valid^*(S', E_p[p], p, s, \langle E_v[v]\rangle_U, \widehat{1}) \wedge valid^*\big(S', E_p[p], cast(p, s, \mathrm{int}), \mathrm{int}, \langle E_v[v]\rangle_U \times \sigma(s), \widehat{0}\big)$$

$$optsize(S', p, s, v) = \mathrm{ite}\Big(E_v[v] = \widehat{0}, \langle M\Big[\Big(\Gamma_\rho\big(cast(p, \mathrm{int}, s)\big), E_p[p], f_{s.\mathrm{len}}\Big)\Big]\rangle_U, \langle E_v[v]\rangle_U\Big)$$

$$mayofint(S', p, s, v) = \exists l \in \mathbb{N}\,.\, l = optsize(S', p, s, v) \wedge valid^*\big(S', E_p[p], cast(p, \mathrm{int}, s), s, l, \widehat{0}\big) \wedge$$
$$valid^*(S', E_p[p], p, \mathrm{int}, l \times \sigma(s), \widehat{1})$$

$$toint(S', p, s, v) = M\big[upd2\big(S', E_p[p], p, cast(p, s, \mathrm{int}), s, \mathrm{int}, \langle E_v[v]\rangle_U\big) \cup$$
$$\big\{\big(\Gamma_\rho(cast(p, s, \mathrm{int}), \widehat{a''}, f_{\mathrm{int.int}}\big) \leftarrow M[(\Gamma_\rho(p'), \widehat{a'}, f_x)]\big|$$
$$\underline{0 \leq i < \langle E_v[v]\rangle_U}, (p', a', a'', f_x) \in fields(\langle E_p[p]\rangle_U + i \times \sigma(s), p, s)\big\}\big]$$

$$ofint(S', p, s, v) = M\big[upd2\big(S', E_p[p], p, cast(p, \mathrm{int}, s), \mathrm{int}, s, \langle E_v[v]\rangle_U\big) \cup$$
$$\big\{(\Gamma_\rho(p'), \widehat{a'}, f_x) \leftarrow M[(\Gamma_\rho(p), \widehat{a''}, f_{\mathrm{int.int}})]\big|$$
$$\underline{0 \leq i < \langle E_v[v]\rangle_U},$$
$$(p, a', a'', f_x) \in fields(\langle E_p[p]\rangle_U + i \times \sigma(s), cast(p, \mathrm{int}, s), s)\big\}\big]$$

Рис. 4.5. Обозначения, используемые в модельной семантике базового языка (окончание)

Предикат $init'(S')$ задает начальные предположения для совокупности массивов M аналогично предикату $init(S)$ в исходной семантике. $lubnd(S', a, s, v)$, $upd'^{v}(S', v, i)$ и $upd'^{p}(S', p, a)$ полностью аналогичны соответствующим обозначениям $lubnd(S, a, s, v)$, $upd^{v}(S, v, i)$ и $upd^{p}(S, p, a)$, используемым в исходной семантике. $upd'^{M}(S', p, f, v)$ обозначает результат обновления состояния памяти в результате записи значения v в поле f структуры, адресуемой указателем p, здесь p — указательная переменная, являющаяся одновременно и указательным термом, имеющим указательный тип с регионом $\Gamma_\rho(p)$. $valid'(S', p, s)$ — условие выделенности модельной памяти для всех полей из множества $\mathbb{F}_{vp}$ (но не $\mathbb{F}_s$) структуры с тегом s, ад-

ресуемой указателем p, а также условие представления значений этих полей в массивах, соответствующих региону $\Gamma_\rho(p)$ и условие интерпретации соответствующих ячеек памяти как полей этой структуры — в отличие от предиката $valid(S, p, f)$ исходной семантики, соответствующего условию выделенности одной ячейки памяти, занимаемой одним указанным полем f из $\mathbb{F}_v \cup \mathbb{F}_p$ в общем массиве B, вне зависимости от интерпретации значения этой ячейки. $valid^*(S', a, p, s, l, v)$ — условие равенства значению v флагов выделенности для всей области памяти, занимаемой массивом структур по адресу a типа s размера l, и представленной массивами, соответствующими регионам указателей, получаемых функцией $\pi(s, p, a)$ (описана в разделе 4.1, рекурсивно перебирает все вложенные структуры) из указателя p; задает также условие интерпретации соответствующих ячеек памяти как полей соответствующих структур (структуры s и вложенных в неё массивов структур). $upd'^M_{f_{valid}}(S', a, p, s, l, v)$ — результат аналогичного условию $valid^*$ обновления значения флагов выделенности на новое значение v. $fresh'(S', a, p, s, l)$ — модельный аналог предиката $fresh(S, a, s, v)$, используемого в $alloc(S, a, p, s, v)$, обеспечивает выполнение аналогичного условия невыделенности всей запрашиваемой для выделения области памяти независимо от регионов и типов структур. $fields(a, p, s)$ — вспомогательная функция, возвращающая для всех полей из $\mathbb{F}_v \cup \mathbb{F}_p$ структуры s по адресу a, доступной через указатель p, и всех транзитивно вложенных в неё массивов структур множество четверок (p, a, a', f), где a — адрес структуры, непосредственно содержащей поле f, p — указатель для определения региона, представляющего значение этого поля, a' — адрес самого поля. $mayalloc'(S', a, p, s, v)$, $mayfree'(S', p, s)$, $alloc'(S', a, p, s, v)$ и $free'(S', p, s)$ — аналоги соответствующих предикатов $mayalloc$, $mayfree$ и результатов обновления состояния памяти $alloc$ и $free$ из исходной семантики. $maytoint(S', p, s, v)$, $mayofint(S', p, s, v)$, $toint(S', p, s, v)$ и $ofint(S', p, s, v)$ — предусловия и ре-

зультаты обновления состояния памяти для формализации модельной семантики инструкций $\mathbf{to\_int}(p, v)$ и $\mathbf{of\_int}(p, s, v)$, осуществляющих переинтерпретацию массива структур (с тегом s или int) размера v, адресуемого указателем p (соответствующего типа $\texttt{pointer}\, s\, \rho$ или $\texttt{pointer}\, \mathrm{int}\, \rho$), в массив структур с тегом int и в массив структур с тегом s соответственно.

Для рефлексивного транзитивного замыкания $\triangleright^*$ отношения $\triangleright$ верна теорема, аналогичная теореме 1 для отношения $\blacktriangleright^*$:

Теорема 2. *Для любой конечной последовательности инструкций oo базового языка и любого начального состояния вычисления $oo\big|S'$ отношение вычисления $\triangleright^*$ однозначно определяет соответствующее непустое подмножество всех возможных значений из множества $\{\top, \bot\}$.*

Будем далее использовать обозначения $oo \triangleright^* \top$ и $oo \triangleright^* \bot$, аналогичные $oo \blacktriangleright^* \top$ и $oo \blacktriangleright^* \bot$.

4.3.1. Корректность модельной семантики

Корректность модельной семантики соответствует утверждению о том, что для любой последовательности инструкций oo выполнено следствие $oo \blacktriangleright^* \bot \implies oo \triangleright^* \bot$. Для доказательства этого утверждения введем отношение $\sim$ для произвольных состояний памяти $S = (B, V, L, E_p, E_v)$ и $S' = (M, E'_p, E'_v)$ в исходной и модельной семантике как показано на рис. 4.7.

Заметим, что при вычислении в исходной семантике для любого состояния памяти выполнен инвариант $\forall a \in \mathbb{B}^d.\ V[a] = 0 \implies L[a] = 0$ (8), а при вычислении в модельной семантике выполнены инварианты $\forall \rho \in \mathbb{R}, a \in \mathbb{B}^d.\ M[(\rho, a, f_{\tau(\rho).\mathrm{valid}})] \in \{\widehat{0}, \widehat{1}\}$ (9) и $\forall \rho \in \mathbb{R}, a \in \mathbb{B}^d.\ M[(\rho, a, f_{\tau(\rho).\mathrm{len}})] \neq \widehat{0} \implies \exists \rho' \in \mathbb{R}.\ M[(\rho', a, f_{\tau(\rho).\mathrm{valid}})] = \widehat{1}$ (10). С помощью этих инвариантов можно проверить следующее утверждение:

rule $\in$

$\{\text{arith}, \text{null}, \text{top}, \text{diff}, \text{bot}, \text{eq}, \text{shift}, \text{field}, \text{container}, \text{assert}, \text{assert}^{\perp}, \text{assume}, \text{assume}^{\top}, \text{mayalias}\}$,

rule' $= [S'/S, init'/init, upd'^{v}/upd^{v}, upd'^{p}/upd^{p}]$rule

deref'

$$\frac{p_1 : s \quad x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad valid'(S', p_1, s)}{x = p_1\text{->}f_x; oo \big| s' \rhd oo \big| upd'^{x}\big(S', x, M\big[\big(\Gamma_{\rho(p_1)}, E_p[p_1], f_x\big)\big]\big)}$$

deref'$^{\perp}$

$$\frac{p_1 : s \quad x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad \neg valid'(S', p_1, s)}{x = p_1\text{->}f_x; oo \big| s' \rhd \perp}$$

assign'

$$\frac{p : s \qquad x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad valid'(S', p_1, s)}{p_1\text{->}f_x = x; oo \big| s' \rhd oo \big| upd'^{M}(S', E_p[p_1], f_x, E_x[x])}$$

assign'$^{\perp}$

$$\frac{p : s \qquad x \in \{p, v\}, f_x \in \mathbb{F}_x \qquad \neg valid'(S', p_1, s)}{p_1\text{->}f_x = x; oo \big| s' \rhd \perp}$$

alloc'

$$\frac{p : s \qquad \exists a \in \mathbb{N}.\ mayalloc'(S', a, p, s, v)}{p = \mathbf{alloc}(v); oo \big| s' \rhd oo \big| alloc'(S', a, p, s, v)}$$

alloc'$^{\top}$

$$\frac{p : s \qquad \forall a \in \mathbb{N}.\ \neg mayalloc'(S', a, p, s, v)}{p = \mathbf{alloc}(v); oo \big| s' \rhd \top}$$

free'

$$\frac{p : s \qquad mayfree'(S', p, s)}{p = \mathbf{free}(p); oo \big| s' \rhd oo \big| free'(S', p, s)}$$

free'$^{\perp}$

$$\frac{p : s \qquad \neg mayfree'(S', p, s)}{p = \mathbf{free}(p); oo \big| s' \rhd \perp}$$

toint'

$$\frac{p : s \qquad maytoint(S', p, s, v)}{p = \mathbf{to\_int}(p, v); oo \big| s' \rhd oo \big| toint(S', p, s, v)}$$

toint'$^{\perp}$

$$\frac{p : s \qquad \neg maytoint(S', p, s, v)}{p = \mathbf{to\_int}(p, v); oo \big| s' \rhd \perp}$$

ofint'

$$\frac{mayofint(S', p, s, v)}{p = \mathbf{of\_int}(p, s, v); oo \big| s' \rhd oo \big| ofint(S', p, s, v)}$$

ofint'$^{\perp}$

$$\frac{\neg mayofint(S', p, s, v)}{p = \mathbf{of\_int}(p, s, v); oo \big| s' \rhd \perp}$$

Рис. 4.6. Модельная семантика базового языка

Теорема 3. *При условии $S \sim S'$ в соответствующих состояниях выполнено следование предусловия каждого правила вывода отношения $\blacktriangleright$ без индекса $\perp$ из предусловия соответствующего правила вывода отношения $\rhd$ (см.*

$$
\begin{aligned}
&S \sim S' \iff \\
&(\forall p \in \mathbb{P}.\ E_p[p] = E'_p[p]) \wedge && (1)\\
&(\forall p \in \mathbb{V}.\ E_v[v] = E'_v[v]) \wedge && (2)\\
&\Big(\forall a \in \mathbb{B}^d. \\
&\quad V[a] = 1 \implies \\
&\qquad \exists \rho \in \operatorname{img} \Gamma_\rho.\ \exists f \in \varphi\big(\tau(\rho)\big) \cup (\mathbb{F}_v \cup \mathbb{F}_p). \\
&\qquad\quad \underline{M[\rho, a \mathbin{\widehat{\frown}} \widehat{o(f)}, f_{\tau(\rho).\mathrm{valid}}] = \widehat{1}}\Big) \wedge && (\underline{3})\\
&\forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \\
&\quad M[(\rho, a, f_{\tau(\rho).\mathrm{valid}})] = \widehat{1} \implies \\
&\qquad \Big(\forall f \in \varphi\big(\tau(\rho)\big) \cap (\mathbb{F}_v \cup \mathbb{F}_p). \\
&\qquad\quad \underline{V[a \mathbin{\widehat{+}} \widehat{o(f)}] = 1} \wedge && (\underline{4})\\
&\qquad\quad \underline{M[(\rho, a, f)] = B[a \mathbin{\widehat{+}} \widehat{o(f)}]} \wedge && (\underline{5})\\
&\qquad\quad \forall \rho' \in \mathbb{R}.\ (\rho \neq \rho' \vee o(f) > 0) \implies \\
&\qquad\qquad \underline{M[\rho', a \mathbin{\widehat{+}} \widehat{o(f)}, f_{\tau(\rho).\mathrm{valid}}] = \widehat{0}}\Big) \wedge && (\underline{6})\\
&\quad M[(\rho, a, f_{\tau(\rho).\mathrm{len}})] \mathbin{\widehat{\times}} \widehat{\sigma(\tau(\rho))} = L[a]. && (7)
\end{aligned}
$$

Рис. 4.7. Определение отношения $\sim$ для произвольных состояний памяти $S = (B, V, L, E_p, E_v)$ и $S' = (M, E'_p, E'_v)$ в исходной и модельной семантике

одноименные с точностью до штрихов правила вывода на рис. 4.3 и 4.6).

Доказательство. Необходимо проверить следующие следствия: $init'(S') \implies init(S)$ — из (3) (от противного) и (8); $\neg init'(S') \implies \neg init(S)$ — из (4) и (10); $valid'(S', p, s) \implies valid(S, p, f)$ — из (4); $\exists a \in \mathbb{N}.\ mayalloc'(S, a, p, s, v) \implies \exists a \in \mathbb{N}.\ mayalloc(S, a, s, v)$ – из (3) (от противного) и (8); $\forall a \in \mathbb{N}.\ \neg mayalloc'(S, a, p, s, v) \implies \forall a \in \mathbb{N}.\ \neg mayalloc(S, a, s, v)$ — из (4); $mayfree'(S', p, s) \implies mayfree(S, p)$ — из (4) и (7); $E_v[v]\,\widehat{\Diamond}\,\widehat{0} \implies E_v[v]\,\widehat{\Diamond}\,\widehat{0}$, $\neg E_v[v]\,\widehat{\Diamond}\,\widehat{0} \implies \neg E_v[v]\,\widehat{\Diamond}\,\widehat{0}$ — из (2); $maytoint(S', p, s, v) \implies \top$, $mayofint(S', p, s, v) \implies \top$, $\top \implies \top$ — тривиально. □

Докажем вспомогательную теорему об отношении $\sim$ для начальных состояний:

Теорема 4. *Для любого начального состояния $S = (B, V, L, E_p, E_v)$, такого что выполнено $init(S)$, существует начальное состояние $S' = (M, E_p, E_v)$ такое, что выполнено $init'(S')$ и $S \sim S'$. Также и для любого $S' = (M, E_p, E_v)$, $init'(S')$ существует $S = (B, V, L, E_p, E_v)$, $init(S)$, такое что $S \sim S'$.*

Доказательство. В качестве S' (соответственно S) можно брать любое начальное состояние заданного вида, для которого выполнено $init'(S')$ (соответственно $init(S)$), так как из условий $init(S)$ и $init(S')$, а также совпадения E_v и E_p непосредственно следует $S \sim S'$ (условия $init(S)$ и $init'(S')$ обращают в ложь посылы обеих импликаций в определении отношения $\sim$). Таким образом, соответствующие состояния S и S' всегда существуют. □

Заметим, что один шаг вычисления как в исходной, так и в модельной семантике удаляет из текущей последовательности инструкций самую первую и

таким образом приводит к одной и той же результирующей последовательности инструкций. Поэтому в формулировках теорем соответствующие результирующие последовательности инструкций можно не различать. Перейдем к следующей вспомогательной теореме о связи между отношениями $\blacktriangleright$ и $\rhd$.

Теорема 5. *Для любых состояний S и S', таких что $S \sim S'$, любых конечных последовательностей инструкций базового языка oo и oo' и любых непустых множеств состояний памяти $\mathfrak{S}$ и $\mathfrak{S}'$ таких, что $\forall S_1 \in \mathfrak{S}.\ \ oo\big|S \blacktriangleright oo'\big|S_1$ и $\forall S'_1 \in \mathfrak{S}'.\ \ oo\big|S' \rhd oo'\big|S'_1$, выполнено $\forall S_1 \in \mathfrak{S}.\, \exists S'_1 \in \mathfrak{S}'.\, S_1 \sim S'_1$ и $\forall S'_1 \in \mathfrak{S}'.\, \exists S_1 \in \mathfrak{S}.\, S_1 \sim S'_1$.*

Доказательство. Полным перебором всех возможных правил вывода рассматриваемых отношений $\blacktriangleright$ и $\rhd$.

Из совпадения исходных и результирующих последовательностей инструкций oo и oo' для обоих рассматриваемых отношений вычисления в условии теоремы следует, что рассматривать следует только соответствующие правила вывода — одноименные правила без штриха для отношения $\blacktriangleright$ и со штрихом (и, возможно, с индексом $\bot$) для отношения $\rhd$, так как разноименные правила вывода применимы только к различающимся в первом элементе последовательностям инструкций. Вначале заметим, что для правил вывода top, bot, deref$^{\bot}$, assign$^{\bot}$, alloc$^{\top}$, free$^{\bot}$, assert$^{\bot}$ и assume$^{\top}$, а также соответствующих им правил вывода со штрихом и правил toint'$^{\bot}$ и ofint'$^{\bot}$ множества состояний $\mathfrak{S}$ и $\mathfrak{S}'$ пусты (не выполнено $oo\big|S \blacktriangleright oo'\big|S_1$ или $oo\big|S' \rhd oo'\big|S'_1$ ни для какого S_1 или S'_1) и, следовательно, утверждение теоремы выполнено тривиально. Для всех остальных правил вывода кроме alloc и alloc' множества $\mathfrak{S}$ и $\mathfrak{S}'$ содержат по одному элементу. Обозначим соответствующие элементы S_1 и S'_1.

Для правил mayalias, assert и assume и соответствующих правил mayalias', assert' и assume' элементы S_1 и S'_1 тривиально совпадают с эле-

ментами S и S' соответственно, что означает выполненность утверждения теоремы. Для правил вывода arith, null, diff, eq, shift, field, container и соответствующих им правил вывода со штрихом отношение $S_1 \sim S'_1$ следует из (1) и (2). Для deref и deref' аналогично отношение $S_1 \sim S'_1$ следует из (1), (2) и (5). Для assign и assign' утверждение $S_1 \sim S'_1$ можно доказать, используя (1), (2), (5), а также (6), чтобы показать (5) для адресов и регионов, отличных от тех, в которых происходит обновление отображения M.

Для free и free' нужно учесть совпадение адресов a в обеих семантиках, следующее из (2), откуда с учетом определения функции $\pi(s, p, a)$ можно показать (3) и (4). Остальные составляющие утверждения $S_1 \sim S'_1$ следуют из определения результатов обновления $free(S, p)$ и $free'(S', p, s)$ и тех же составляющих для отношения $S \sim S'$. В случае правил toint, toint', ofint и ofint' для доказательства используются все составляющие утверждения $S \sim S'$, кроме (7), а также определения соответствующих результатов обновления и вспомогательной функции $fields(a, p, s)$. Для оставшихся правил alloc и alloc' соответствующие состояния во множествах $\mathfrak{S}$ и $\mathfrak{S}'$ могут быть найдены с помощью функции $fields(a, p, s)$. Для каждой четверки (p, a, a', f) из множества $fields(a, p, s)$ для соответствующих состояний $S_1 \sim S'_1$ значение $B[\widehat{a'}]$ совпадает со значением $M[(\Gamma_\rho(p), \widehat{a}, f)]$, что дает (5), а все остальные значения при обновлении в правилах alloc и alloc' определены однозначно и с учетом (8) и определения результата обновления обеспечивают (3), (4), (6) и (7). □

Для произвольных состояний вычисления S_v в исходной и S'_v в модельной семантике введем расширенное отношение $\sim$ следующим образом: $S_v \sim S'_v$ тогда и только тогда, когда либо $S_v = \top$, либо $S'_v = \bot$, либо $S_v = oo\big|S$, $S'_v = oo\big|S'$ и $S \sim S'$. Докажем следующую теорему, аналогичную теореме 5, но формулируемую для произвольных состояний вычисления, включая значения $\top$ и $\bot$:

Теорема 6. *Для любых состояний S и S', таких что $S \sim S'$, любых конечных последовательностей инструкций базового языка oo и oo' и любых множеств состояний вычисления $\mathfrak{S}_\mathfrak{v}$ и $\mathfrak{S}'_\mathfrak{v}$ таких, что $\forall S_v \in \mathfrak{S}_\mathfrak{v}\,.\; oo\big|S \blacktriangleright S_v$ и $\forall S'_v \in \mathfrak{S}'_\mathfrak{v}\,.\quad oo\big|S' \rhd S'_v$, выполнено $\forall S_v \in \mathfrak{S}_\mathfrak{v}\,.\quad \exists S'_v \in \mathfrak{S}'_\mathfrak{v}\,.\quad S_v \sim S'_v$ и $\forall S'_v \in \mathfrak{S}'_\mathfrak{v}\,.\, \exists S_v \in \mathfrak{S}_\mathfrak{v}\,.\, S_v \sim S'_v$.*

Доказательство. Для начала заметим, что в силу непересечения правил вывода как по первым инструкциям последовательности, так и по предусловиям (предусловия дополняющих правил с индексами $\top$ и $\bot$ являются отрицаниями предусловий соответствующих правил без индексов) для обоих отношений вычисления, каждое из множеств $\mathfrak{S}_\mathfrak{v}$ и $\mathfrak{S}'_\mathfrak{v}$ может содержать либо один элемент $\top$, либо один элемент $\bot$, либо множество состояний вычисления вида $oo\big|S$ с различными состояниями памяти, как в условии теоремы 5.

Рассмотрим теперь все возможные случаи для множеств состояний вычисления $\mathfrak{S}_\mathfrak{v}$ и $\mathfrak{S}'_\mathfrak{v}$. Для случаев $\mathfrak{S}_\mathfrak{v} = \{\top\}$ и $\mathfrak{S}'_\mathfrak{v} = \{\bot\}$ утверждение теоремы выполнено по определению расширенного отношения $\sim$. В случае $\mathfrak{S}_\mathfrak{v} = \left\{S_1 \middle| oo|S \blacktriangleright oo'|S_1\right\}$ и $\mathfrak{S}'_\mathfrak{v} = \left\{S'_1 \middle| oo|S' \blacktriangleright oo'|S'_1\right\}$ выполнены условия теоремы 5 и требуемое утверждение совпадает с утверждением этой теоремы. Случаи $\mathfrak{S}_\mathfrak{v} = \left\{S_1 \middle| oo|S \blacktriangleright oo'|S_1\right\}$ при $\mathfrak{S}'_\mathfrak{v} = \{\top\}$, $\mathfrak{S}_\mathfrak{v} = \{\bot\}$ при $\mathfrak{S}'_\mathfrak{v} = \left\{S'_1 \middle| oo|S' \blacktriangleright oo'|S'_1\right\}$ и $\mathfrak{S}_\mathfrak{v} = \{\bot\}$ при $\mathfrak{S}'_\mathfrak{v} = \{\top\}$ невозможны вследствие теоремы 3, так как в этих случаях из предусловий правил вывода для отношения $\rhd$ без индекса $\bot$ следуют предусловия для соответствующих правил вывода отношения $\blacktriangleright$, по которым, в свою очередь однозначно (в силу непересечения по предусловиям) выводятся состояния вычисления, не соответствующие требуемым ($\{\top\}$ вместо множества с состояниями памяти, множество с состояниями памяти вместо $\{\bot\}$ и $\{\top\}$ вместо $\{\bot\}$ соответственно). □

Расширим теперь определение отношения $\sim$ на множества состояний вычисления в соответствии со следствием теоремы 6 следующим обра-

зом: $\mathfrak{S}_\mathfrak{v} \sim \mathfrak{S}'_\mathfrak{v} \iff (\forall S_v \in \mathfrak{S}_\mathfrak{v}.\ \exists S'_v \in \mathfrak{S}'_\mathfrak{v}.\ S_v \sim S'_v) \wedge (\forall S'_v \in \mathfrak{S}'_\mathfrak{v}.\ \exists S_v \in \mathfrak{S}_\mathfrak{v}.\ S_v \sim S'_v)$. Из теорем 4 и 6 в силу сохранения расширенного отношения $\sim$ при объединении соответствующих множеств для любых начальных состояний S, $init(S)$ и S', $init'(S')$ и любых последовательностей инструкций oo по индукции получаем $\mathfrak{S}_\mathfrak{v} \sim \mathfrak{S}'_\mathfrak{v}$ для всех $\mathfrak{S}_\mathfrak{v}, \mathfrak{S}'_\mathfrak{v} \subseteq \{\top, \bot\}$, таких что $oo \big| S \blacktriangleright^* \mathfrak{S}_\mathfrak{v}$ и $oo \big| S' \triangleright^* \mathfrak{S}'_\mathfrak{v}$. Отсюда по определению расширенного отношения $\sim$ непосредственно следует основная теорема этого подраздела:

Теорема 7 (Корректность модельной семантики)**.** *Для любой конечной последовательности инструкций базового языка oo выполнено следствие* $oo \blacktriangleright^* \bot \implies oo \triangleright^* \bot$.

4.3.2. Полнота модельной семантики

Введем дополнительное ограничение на контекст типизации Γ. Пусть существуют отображения Δ и Δ^{-1}, где $\Delta, \Delta^{-1} \in \mathbb{R} \times \mathbb{F}_s \to \mathbb{R}$, для которых выполнено:

$$
\begin{aligned}
&\forall p \in \mathcal{T}^*. \\
&\quad \Big(\forall f_s \in \varphi\Big(\tau\big(\Gamma_\rho(p)\big)\Big) \cap \mathbb{F}_s. \\
&\qquad \Gamma_\rho(\texttt{\&}p\texttt{->}f_s) = \Delta\big(\Gamma_\rho(p), f_s\big) \wedge \\
&\qquad \Delta^{-1}\Big(\Delta\big(\Gamma_\rho(p), f_s\big), f_s\Big) = \Gamma_\rho(p)\Big) \wedge \qquad (11) \\
&\quad \Big(\forall c \in \mathbb{N}\backslash\{0\}, f_{s'} : \texttt{struct}\,\tau\big(\Gamma_\rho(p)\big)\ \texttt{[}c\texttt{]} \in \mathbb{F}_s. \\
&\qquad \Gamma_\rho\big(\texttt{container\_of}(p, f_{s'})\big) = \Delta^{-1}\big(\Gamma_\rho(p), f_{s'}\big) \wedge \\
&\qquad \Delta\Big(\Delta^{-1}\big(\Gamma_\rho(p), f_{s'}\big), f_{s'}\Big) = \Gamma_\rho(p)\Big).
\end{aligned}
$$

Это условие по сути задает достаточное ограничение на возможные виды анализов регионов, совместимые с представленной модельной семантикой при

$$\mathbb{F}_d \subseteq \mathbb{F}_v \cup \mathbb{F}_p, \quad \mathbb{P}_d \cap \mathbb{P} = \emptyset, \quad x \in \{p, v\}, \quad v_a \notin \mathbb{V},$$
$$\mathbb{S}' = \mathbb{S} \cup \{s_{x_d} \big| f_{x_d} \in \mathbb{F}_d\}, \quad \mathbb{F}'_v = \mathbb{F}_v, \quad \mathbb{F}'_p = \mathbb{F}_p, \quad \mathbb{F}'_s = \mathbb{F}_s \cup \{f_{s_{x_d}} \big| f_{x_d} \in \mathbb{F}_d\},$$
$$\mathbb{P}' = \mathbb{P} \cup \mathbb{P}_d, \quad \mathbb{V}' = \mathbb{V} \cup \{v_a\}, \quad \mathbb{R}' \supseteq \mathbb{R},$$
$$\forall s \in \mathbb{S}\,.\ \varphi'(s) = \varphi(s) \backslash \mathbb{F}_d \cup \{f_{s_{x_d}} \big| f_{x_d} \in \varphi(s) \cap \mathbb{F}_d\}, \quad \forall s_{x_d} \in \mathbb{S}' \backslash \mathbb{S}\,.\ \varphi(s_{x_d}) = \{f_{x_d}\},$$
$$\forall f \in \operatorname{dom} \Phi\,.\ \Phi'(f) = \Phi(f), \quad f_{s_{x_d}} : \texttt{struct}\ s_{x_d}\ \texttt{[1]}, \quad \operatorname{dom} \Gamma' \supseteq \operatorname{dom} \Gamma,$$
$$\forall f_{x_d} \in \mathbb{F}_d\,.\ p_{x_d} : \texttt{pointer}\ s_{x_d}\ \rho \in \mathbb{P}_d,$$
$$oo, oo' : \texttt{unit} \in \mathcal{T}, \quad f_{v_d} \in \mathbb{F}_d, \quad f_{p_d} \in \mathbb{F}_p, \quad f_{s_{v_d}}, f_{s_{p_d}} \in \mathbb{F}'_s \backslash \mathbb{F}_s,$$
$$s \in \mathbb{S}', \quad p, p_1, p_2 \in \mathbb{P}', \quad v \in \mathbb{V}', \quad p_d, p_{v_d}, p_{p_d} \in \mathbb{P}_d$$

$$oo \ \dot{\sim} \ oo$$

$$oo \dot{\sim} oo' \implies v = p\texttt{->}f_{v_d}; \ oo \ \dot{\sim} \ p_{v_d} = \&p\texttt{->}f_{s_{v_d}}; \ v = p_{v_d}\texttt{->}f_{v_d}; \ oo'$$
$$oo \dot{\sim} oo' \implies p_1 = p_2\texttt{->}f_{p_d}; \ oo \ \dot{\sim} \ p_{p_d} = \&p_2\texttt{->}f_{s_{p_d}}; \ p_1 = p_{p_d}\texttt{->}f_{p_d}; \ oo'$$
$$oo \dot{\sim} oo' \implies p\texttt{->}f_{v_d} = v; \ oo \ \dot{\sim} \ p_{v_d} = \&p\texttt{->}f_{s_{v_d}}; \ p_{v_d}\texttt{->}f_{v_d} = v; \ oo'$$
$$oo \dot{\sim} oo' \implies p_1\texttt{->}f_{p_d} = p_2; \ oo \ \dot{\sim} \ p_{p_d} = \&p_1\texttt{->}f_{s_{p_d}}; \ p_{p_d}\texttt{->}f_{p_d} = p_2; \ oo'$$
$$oo \dot{\sim} oo' \implies oo \ \dot{\sim} \ p_d = t_p; \ oo'$$
$$oo \dot{\sim} oo' \implies oo \ \dot{\sim} \ v_a = i; \ oo'$$
$$oo \dot{\sim} oo' \implies oo \ \dot{\sim} \ \textbf{to\_int}(p, v); \ oo'$$
$$oo \dot{\sim} oo' \implies oo \ \dot{\sim} \ \textbf{of\_int}(p, s, v); \ oo'$$
$$oo \dot{\sim} oo' \implies oo \ \dot{\sim} \ \textbf{may\_alias}(p_1, p_2); \ oo'$$

Рис. 4.8. Определение отношения $\dot{\sim}$

условии сохранения её полноты. Далее будем доказывать полноту модельной семантики при условии выполненности этого ограничения.

Полнотой модельной семантики назовем утверждение о том, что существует некоторое однозначное преобразование $\mathcal{C}$ конечных корректно типизированных последовательностей инструкций базового языка, не содержащих инструкции $\textbf{to\_int}(p, v)$ и $\textbf{of\_int}(p, s, v)$ (будем обозначать соответствующее множество последовательностей через $\mathcal{T}$), включающее также соответствующее преобразование контекстов этих последовательностей, такое, что $\forall oo \in \mathcal{T}, r \in \{\top, \bot\}\,.\ oo \blacktriangleright^* r \iff \mathcal{C}(oo) \rhd^* r$. Так как на практике, как правило, с целью сохранения эффективности использования модельной семан-

тики для представления операций с указателями в виде логических формул используется другое определение полноты, а именно утверждение о том, что преобразование $\mathcal{C}$, такое что $\forall r \in \{\top, \bot\}\,.\; oo \blacktriangleright^* r \iff \mathcal{C}(oo) \rhd^* r$, существует отдельно для каждой исходной последовательности инструкций, вместо одного конкретного определения $\mathcal{C}$ будем рассматривать класс преобразований, заданный параметризованным отношением $\dot{\sim}$ между исходными и преобразованными последовательностями. Затем укажем значения параметров $\mathbb{F}_d$ и $\mathbb{P}_d$ и дополнительные условия, при которых которых отношение $\dot{\sim}$ однозначно задает искомое преобразование $\mathcal{C}$. На практике преобразование $\mathcal{C}$ определяется для каждой программы в отдельности в соответствии с задаваемыми пользователем аннотациями. Однако все такие преобразования удовлетворяют соответствующему аналогу условия $oo \dot{\sim} \mathcal{C}(oo)$ для соответствующих базовых путей. Таким образом, на практике полнота модельной семантики гарантирует существование для каждой отдельно взятой программы некоторого набора аннотаций, позволяющего сделать исходную и модельную семантики выполнения этой программы полностью эквивалентными с точки зрения результата верификации.

Отношение $\dot{\sim}$, определенное на рис. 4.8, является отношением между исходными последовательностями инструкций и последовательностями, в которых выполнены произвольные преобразования одного из следующих типов:

- введение дополнительных указательных переменных из множества $\mathbb{P}_d \not\subseteq \mathbb{P}$;

- выделение поля $f_{x_d} \in \mathbb{F}_d \subseteq \mathbb{F}_v \cup \mathbb{F}_p$ одной из структур $s \in \mathbb{S}$ в отдельную структуру $s_{x_d} \notin \mathbb{S}$, $\varphi(s_{x_d}) = \{f_{x_d}\}$ с соответствующей заменой поля f_{x_d} на поле $f_{s_{x_d}}$: $\texttt{struct}\, s_{x_d}\, \texttt{[1]}$, а всех инструкций, обращающихся к полю f_{x_d}, на исходно-семантически (то есть относительно $\blacktriangleright$) эквивалентные подпоследовательности из двух соответствующих инструкций

(см. рис. 4.8);

- вставка инструкций присваивания $p_d = t_p$ во вновь введенные указательные переменные $p_d \in \mathbb{P}_d$ в произвольные позиции;
- добавление во множество $\mathbb{V}$ одной ранее не принадлежащей $\mathbb{V}$ целочисленной переменной v_a;
- добавление инструкций присваивания в новую целочисленную переменную v_a;
- вставка произвольных инструкций $\mathbf{to\_int}(p, v)$, $\mathbf{of\_int}(p, s, v)$ и $\mathbf{may\_alias}(p_1, p_2)$ в произвольные позиции.

Заметим, что преобразование инструкций, обращающихся к полям $f_{x_d} \in \mathbb{F}_d$ обязательно для любого преобразования $\mathcal{C}$, такого что $oo \dot{\sim} \mathcal{C}(oo)$ для $oo \in \mathcal{T}$, так как результирующая программа является корректно типизированной. Докажем для отношения $\dot{\sim}$ следующую вспомогательную теорему:

Теорема 8. *Для любого корректного преобразования последовательностей инструкций базового языка и соответствующих контекстов $\mathcal{C}$ и любой конечной последовательности инструкций базового языка $oo \in \mathcal{T}$ выполнено* $oo \dot{\sim} \mathcal{C}(oo) \implies \forall r \in \{\top, \bot\} \,.\, oo \blacktriangleright^* r \iff \mathcal{C}(oo) \blacktriangleright^* r$.

Доказательство. Утверждение теоремы следует из исходной семантики инструкций $\mathbf{to\_int}(p, v)$, $\mathbf{of\_int}(p, s, v)$ и $\mathbf{may\_alias}(p_1, p_2)$ (имеют семантику пустых инструкций, см. рис. 4.3), условия $\mathbb{P}_d \not\subseteq \mathbb{P}$ (гарантирует, что присваивания $p_d = t_p$ выполняются в неиспользуемые переменные), а также эквивалентности соответствующих подпоследовательностей из одной и двух инструкций, соответствующих обращениям к полям $f_{x_d} \in \mathbb{F}_d$ в последовательностях oo и $\mathcal{C}(oo)$ (легко проверяется по правилам вывода отношения $\blacktriangleright$). □

Из приведенной вспомогательной теоремы следует, что для доказательства полноты модельной семантики достаточно доказать утверждение $\forall r \in \{\top, \bot\}.\ \mathcal{C}(oo) \blacktriangleright^* r \iff \mathcal{C}(oo) \rhd^* r$ для некоторого преобразования $\mathcal{C}$. Это утверждение будем получать по индукции аналогично тому, как было получено утверждению теоремы о корректности, но в применении к преобразованной последовательности инструкций. Для этого вместо расширенного отношения $\sim$ рассмотрим его усиление $\approx$, определенное следующим образом: для произвольных состояний вычисления S_v в исходной и S'_v в модельной семантике $S_v \approx S'_v$ тогда и только тогда, когда либо $S_v = \top$ и $S'_v = \top$, либо $S_v = \bot$ и $S'_v = \bot$, либо $S_v = oo\big|S$, $S'_v = oo\big|S'$ и $S \sim S'$. Для доказательства по индукции достаточно получить соответствующий аналог теоремы 6 для отношения $\approx$ для преобразованных последовательностей инструкций. Учитывая определение отношения $\sim$ и теорему 6, для этого достаточно доказать для произвольных состояний памяти S и S', $S \sim S'$, в исходной и модельной семантике соответственно и последовательностей инструкций oo и oo', удовлетворяющих некоторым дополнительным ограничениям, два утверждения: $oo\big|S \blacktriangleright \top \implies oo\big|S' \rhd \top$ и $oo\big|S \blacktriangleright oo'\big|S_1 \implies oo\big|S' \rhd oo'\big|S'_1$ для любого S_1 и некоторого S'_1. Первое утверждение верно вследствие следующей теоремы:

Теорема 9. *При условии $S \sim S'$ в соответствующих состояниях выполнены следствия предусловий каждого правила вывода отношения $\rhd$ с индексом $\top$ и правила top' из предусловий соответствующих правил вывода отношения $\blacktriangleright$.*

Доказательство. $init(S) \implies init'(S')$ – из инварианта (10) и условия (4) в определении $\sim$; $\forall a \in \mathbb{N}.\ \neg mayalloc(S, a, s, v) \implies \forall a \in \mathbb{N}.\ \neg mayalloc'(S', a, p, s, v)$ — из условия (3); $\neg E_v[v]\,\widehat{\Diamond}\,\widehat{0} \implies \neg E_v[v]\,\widehat{\Diamond}\,\widehat{0}$ — из условия (2). □

Аналогичная теорема для второго утверждения ($oo\big|S \blacktriangleright oo'\big|S_1 \implies$

$oo \big| S' \rhd oo' \big| S'_1$) требует применения к последовательности инструкций определенных преобразований.

Теорема 10. *Пусть далее все* $p_{di_j} \in \mathbb{P}_d$ *различны. Возьмем* $\mathbb{F}_d = \mathbb{F}_v \cup \mathbb{F}_p$ *и рассмотрим следующее преобразование последовательности инструкций* $\mathcal{C}$*: сначала применим выделение всех полей из* $\mathbb{F}_d$ *в отдельные структуры согласно определению отношения* $\dot{\sim}$*; затем после каждой* i*-ой по порядку в исходной последовательности инструкции вида* $p = \mathbf{alloc}(v)$*, где* p : `pointer` $s\,\rho$*, вставим последовательность из трех инструкций "*$\mathbf{to\_int}(p, v)$; p_{di_1} = `&`p`->`$f_{s.\mathrm{void}}$; $p_{di_2} = \mathbf{container\_of}(p_{di_1}, f_{\mathrm{int.void}})$*", перед каждой* i*-ой по порядку инструкцией, содержащей обращение к полю* $f_d \in \mathbb{F}_d$ *структуры по некоторой указательной переменной* p : `pointer` $s\,\rho$*, вставим последовательность из четырех инструкций "*$p_{di_1} =$ `&`p`->`$f_{s.\mathrm{void}}$; $p_{di_2} = \mathbf{container\_of}(p_{di_1}, f_{\mathrm{int.void}})$; $v = 1$; $\mathbf{of\_int}(p_{di_2}, s, v)$*", после каждой такой* i*-ой инструкции вставим инструкцию* $\mathbf{to\_int}(p, v)$*, а перед каждой* i*-ой по порядку инструкцией* $\mathbf{free}(p)$*, где* p : `pointer` $s\,\rho$*, вставим последовательность четырех инструкций "*$p_{di_1} =$ `&`p`->`$f_{s.\mathrm{void}}$; $p_{di_2} = \mathbf{container\_of}(p_{di_1}, f_{\mathrm{int.void}})$; $v = 0$; $\mathbf{of\_int}(p_{di_2}, s, v)$*"; наконец, выберем среди всех вновь введенных переменных* p_{di_2} *для инструкций вида* $p = \mathbf{alloc}(v)$ *произвольную переменную* p_{du} *и добавим в конец последовательности инструкции вида* $\mathbf{may\_alias}(p_{di_2}, p_{du})$ *для каждой введенной переменной* p_{di_2}*. Если выполнено ограничение* (11)*, то при вычислении последовательности инструкций* $\mathcal{C}(oo)$ *для исходной последовательности инструкций* oo*, не содержащей инструкции* $\mathbf{to\_int}(p, v)$ *и* $\mathbf{of\_int}(p, s, v)$*, при условии* $S \sim S'$ *в соответствующих состояниях выполнены следствия предусловий для правил вывода deref', assign', free' и assert' отношения* $\rhd$ *из предусловий соответствующих правил вывода отношения* $\blacktriangleright$*, а также следствия всех предусловий для инструкций, вставленных при выполнении преобразования* $\mathcal{C}$*, из предусло-*

вий либо результатов обновления состояния памяти соответствующих правил вывода отношения $\blacktriangleright$ *без символа* $\bot$ *для соответствующих непосредственно смежных с ними инструкций (либо следствие из предусловия следующей инструкции, либо следствие из результата обновления предыдущей).*

Доказательство. Следует из условий (1), (2) и (3) в определении отношения $\sim$ с учетом описанных преобразований и ограничения (11) (либо в (3) достаточно подставить $\rho = \Gamma_\rho(p_{d_u})$ и $f = f_{\text{int.int}}$, либо соответствующее предусловие следует из результата обновления памяти в непосредственно предшествующей операции). □

Из теорем 9 и 10 следует следующая теорема о полноте модельной семантики:

Теорема 11 (Полнота модельной семантики)**.** *Если выполнено ограничение* (11)*, то существует однозначное преобразование* $\mathcal{C}$ *конечных корректно типизированных последовательностей инструкций базового языка, не содержащих инструкции* $\mathbf{to\_int}(p, v)$ *и* $\mathbf{of\_int}(p, s, v)$*, включающее также соответствующее преобразование контекстов этих последовательностей, такое, что* $\forall oo \in \mathcal{T}, r \in \{\top, \bot\}\,.\; oo \blacktriangleright^* r \iff \mathcal{C}(oo) \rhd^* r$*.*

4.4. Анализ регионов для базового языка с поддержкой вложенности

Анализ регионов для базового языка накладывает дополнительные ограничения на контекст типизации его термов. Эти ограничения всегда имеют вид равенств между регионами указателей на структуры с одинаковыми тегами. Введение таких ограничений позволяет уменьшить количество задаваемых пользователем спецификаций, соответствующих инструкциям вида

$$\text{diff}^{\circ}\ \frac{P_p[p_1] = P_p[p_2]}{v = p_1 - p_2;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p, P_B} \qquad \text{diff}^{\circ\perp}\ \frac{P_p[p_1] \neq P_p[p_2]}{v = p_1 - p_2;\, oo\big|S, P_p, P_B \blacktriangleright \perp}$$

$$\text{eq}^{\circ}\ \frac{P_p[p_1] = P_p[p_2] \vee V\big[E_p[p_1]\big] \wedge V\big[E_p[p_2]\big]}{v = p_1 == p_2;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p, P_B} \qquad \text{eq}^{\circ\perp}\ \frac{P_p[p_1] \neq P_p[p_2] \qquad \neg V\big[E_p[p_1]\big] \vee \neg V\big[E_p[p_2]\big]}{v = p_1 == p_2;\, oo\big|S, P_p, P_B \blacktriangleright \perp}$$

$$\text{shift}^{\circ}\ p_1 = p_2 + v;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p\big[p_1 \leftarrow P_p[p_2]\big], P_B \qquad \text{toint}^{\circ}\ \mathbf{to\_int}(p, v);\, oo\big|S, P_p, P_B \blacktriangleright \perp$$

$$\text{field}^{\circ}\ p_1 = \&p_2\texttt{->}f_s;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p\big[p_1 \leftarrow P_p[p_2]\|[o(f_s)]\big], P_B \qquad \text{ofint}^{\circ}\ \mathbf{of\_int}(p, s, v);\, oo\big|S, P_p, P_B \blacktriangleright \perp$$

$$\text{container}^{\circ}\ \frac{\exists l \in \mathcal{P}\,.\, P_p[p_2] = l\|[o(f_{s'})]}{p_1 = \mathbf{container\_of}(p_2, f_{s'});\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p[p_1 \leftarrow l], P_B}$$

$$\text{container}^{\circ\perp}\ \frac{\forall l \in \mathcal{P}\,.\, P_p[p_2] \neq l\|[o(f_{s'})]}{p_1 = \mathbf{container\_of}(p_2, f_{s'});\, oo\big|S, P_p, P_B \blacktriangleright \perp} \qquad \text{null}^{\circ}\ p = \text{NULL};\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p[p \leftarrow \perp], P_B$$

$$\text{derefv}^{\circ}\ \frac{P_p[p] \neq \perp}{v = p\texttt{->}f_v;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p, P_B} \qquad \text{derefv}^{\circ\perp}\ \frac{P_p[p] = \perp}{v = p\texttt{->}f_v;\, oo\big|S, P_p, P_B \blacktriangleright \perp}$$

$$\text{derefp}^{\circ}\ \frac{P_p[p] \neq \perp}{p_1 = p_2\texttt{->}f_p;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p\big[p_1 \leftarrow P_B\big[E_p[p_2] \mathbin{\widehat{+}} \widehat{o(f_p)}\big]\big], P_B} \qquad \text{derefp}^{\circ\perp}\ \frac{P_p[p] = \perp}{p_1 = p_2\texttt{->}f_p;\, oo\big|S, P_p, P_B \blacktriangleright \perp}$$

$$\text{assignv}^{\circ}\ \frac{P_p[p] \neq \perp}{p\texttt{->}f_v = v;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p, P_B[E_p[p] \mathbin{\widehat{+}} \widehat{o(f_p)} \leftarrow \perp]} \qquad \text{assignv}^{\circ\perp}\ \frac{P_p[p] = \perp}{p\texttt{->}f_v = v;\, oo\big|S, P_p, P_B \blacktriangleright \perp}$$

$$\text{assignp}^{\circ}\ \frac{P_p[p_1] \neq \perp}{p_1\texttt{->}f_p = p_2;\, oo\big|S, P_p, P_B \blacktriangleright oo\big|S', P_p, P_B\big[E_p[p_1] \mathbin{\widehat{+}} \widehat{o(f_p)} \leftarrow P_p[p_2]\big]}$$

Рис. 4.9. Дополнительные правила вывода для отношения $\blacktriangleright$ (начало).

$\mathbf{may\_alias}(p_1, p_2)$, необходимым для верификации каждой конкретной программы (всех последовательностей инструкций на базовых путях). Предлагаемый далее подход к анализу регионов не является полным в общем случае, но так как добавление произвольных инструкций вида $\mathbf{may\_alias}(p_1, p_2)$ не нарушает условий полноты модельной семантики, при использовании анализа регионов для каждой программы по прежнему продолжает существовать неко-

$$pads(s) = \bigcup_{f \in \varphi(s)} \begin{cases} \{o(f)\},\ f \in \mathbb{F}_c, \\ \emptyset,\ f \in (\mathbb{F}_v \cup \mathbb{F}_p) \backslash \mathbb{F}_c, \\ \{o(f) + i \times \sigma(s') + o' \big| \\ \quad 0 \le i < c, o' \in pads(s')\},\ f : \texttt{struct}\, s'\, \texttt{[}c\texttt{]} \in \mathbb{F}_s \end{cases} \quad \begin{array}{l} init(S, P_p, P_B) = init(S) \wedge P_p = \{p \mapsto \bot \big| p \in \mathbb{P}\} \wedge \\ V : \mathbb{B}^d \to \{0,1,2\} \qquad P_B = \{a \mapsto \bot \big| a \in \mathbb{B}^d\} \end{array}$$

$$exclpads(S, p, s) = (B, V\left[\left\{E_p[p] \mathbin{\widehat{+}} \widehat{o} \leftarrow 2 \big| o \in pads(s)\right\}\right], L, E_p, E_v)$$

$$allocated(S, p, l) = \forall i \in \mathbb{N}.\ i < l \wedge \Big(\forall o \in {}_{pads}\Big(\tau\big(\Gamma_\rho(p)\big)\Big).\ i \neq o)\Big) \Rightarrow V\left[E_p[p] \mathbin{\widehat{+}} \widehat{i}\right] = 1$$

$$\frac{P_p[p_1] = \bot}{p_1\texttt{->}f_p = p_2; {}_{oo}\big|S, P_p, P_B \blacktriangleright \bot}\ \text{assignp}^{\circ\bot} \qquad \frac{p : \texttt{pointer}\, s\, \rho \qquad a^* \text{ — новая переменная}}{p = \mathbf{alloc}(v); {}_{oo}\big|S, P_p, P_B \blacktriangleright {}_{oo}\big|exclpads(S', p, s), P_p[p \leftarrow [a^*]], P_B}\ \text{alloc}^{\circ}$$

$$\frac{P_p[p] \neq \bot}{p = \mathbf{free}(p); {}_{oo}\big|S, P_p, P_B \blacktriangleright {}_{oo}\big|S', P_p, P_B}\ \text{free}^{\circ} \qquad \frac{P_p[p] = \bot}{p = \mathbf{free}(p); {}_{oo}\big|S, P_p, P_B \blacktriangleright \bot}\ \text{free}^{\circ\bot}$$

Рис. 4.10. Дополнительные правила вывода для отношения $\blacktriangleright$ (окончание).

торый набор дополнительных аннотаций, позволяющих сделать исходную и модельную семантики эквивалентными с точки зрения результата верификации. Предлагаемый подход к автоматическому выводу регионов сформулируем в виде дополнительных ограничений на контекст типизации термов базового языка Г. Формулируемые ограничения являются тривиально разрешимыми для любой входной последовательности инструкций, так как содержат только равенства между регионами указателей на структуры с одинаковыми тегами, и, таким образом, всегда допускают тривиальное решение, присваивающее всем указательным термам с одинаковым тегом структуры один и тот же регион. На практике, однако, предлагаемые ограничения позволяют разработать соответствующий алгоритм анализа регионов на основе структуры непересекающихся множеств, помещающий все указательные термы, для которых отсутствуют соответствующие ограничения, в различные регионы. Таким образом, на практике анализ регионов изначально работает в оптимистических (сильных) предположениях, которые могут быть ослаблены с помощью задаваемых пользователем аннотаций.

Следует отметить, что на практике в инструменте дедуктивной верификации анализ регионов должен поддерживать процедурную абстракцию (определения и вызовы функций для языка Си), а также обеспечивать конечность числа выводимых регионов для поддержки рекурсивных функций, индуктивных предикатов и кванторов, связывающих указательные переменные. Требование конечности числа регионов может быть легко обеспечено с помощью принудительного приравнивания (унификации) регионов для потенциально возможных термов, явно не присутствующих в исходном коде программы (возникают, например, при использовании связанных кванторами указательных переменных, рекурсии, индуктивных предикатов). Для вызовов функций практическая реализация анализа регионов распространяет ограничения на контекст типизации только в одном направлении — из вызываемой функции в вызывающую. При таком подходе возможны случаи, при которых возникает необходимость разбиения некоторых регионов вызывающей функции в некоторых точках вызова. В таких случаях иногда возможно усилить предположения анализа регионов за счет проверки непересечения областей памяти, адресуемых в вызываемой функции через указательные термы с разными регионами, которые при вызове сопоставляются с одним и тем же регионом взывающей функции. Для этого вызываемая функция должна быть снабжена соответствующими (проверяемыми с помощью условий верификации) аннотациями, задающими адресуемые функцией области памяти. Проверка непересечения областей памяти позволяет восстановить инварианты, обеспечивающие корректность модельной семантики (условие (6) в определении отношения $\sim$ в разделе 4.3), при переходе к новой совокупности массивов M. Так как семантика вызовов функций в инструменте верификации моделируется с помощью соответствующих им контрактов, для трансляции вызова функции, требующей разбиения некоторых регионов, достаточно транслировать соответствующий контракт в контексте вызывающей

функции. Семантически результат такой трансляции будет эквивалентен копированию разбиваемого региона в каждый из соответствующих регионов вызываемой функции непосредственно перед её вызовом с последующим слиянием возможно модифицированных регионов, однозначно определенным из условий непересечения адресуемых областей памяти. Проверка непересечения областей памяти с учетом условия (6) в определении отношения $\sim$ может быть сведена к проверке непересечения соответствующих множеств адресов. В случае необходимости разбиения регионов при отсутствии необходимых аннотаций в вызываемой функции инструмент верификации завершается с соответствующим сообщением об ошибке.

Для описания предлагаемого подхода к анализу регионов сформулируем соответствующие ограничения на контекст типизации Γ, а затем сформулируем условия, при которых указанные ограничения обеспечивают полноту модельной семантики. Итак, пусть помимо ограничений, связанных с существованием отображения τ (см. рис. 4.4 в разделе 4.3) и отображений Δ и Δ^{-1}, удовлетворяющих условию (11) (раздел 4.3), для контекста типизации Γ выполнено также условие существования отображения $\Delta' : \mathbb{R} \times \mathbb{F}_p \to \mathbb{R}$, такого что

$$\begin{gathered}\forall p \in \mathcal{T}^* \mathbf{.}\ \forall f_p \in \varphi\Big(\tau\big(\Gamma_\rho(p)\big)\Big) \cap \mathbb{F}_p \mathbf{.} \\ \Gamma_\rho(p\text{->}f_p) = \Delta'\big(\Gamma_\rho(p), f_p\big).\end{gathered} \tag{12}$$

Рассмотрим теперь исходные последовательности инструкций, для которых перед каждым полем-массивом структур f_s в каждой соответствующей структуре присутствует $n_{f_s} \in \mathbb{N}$ не используемых (не адресуемых) в инструкциях последовательности целочисленных полей, в совокупности составляющих множество $\mathbb{F}_c$. Рассмотрим также отношение вычисления $\blacktriangleright\!\!\!\bigcirc$, изменяющее исходное отношение $\blacktriangleright$ путем введения дополнительных отображений P_p и P_B и дополнительных проверочных правил вывода, приведенных на рис. 4.9 и 4.10.

Проверочные правила вывода применяются после применения какого-либо правила вывода для отношения $\blacktriangleright$ в случае, когда получившееся при этом состояние вычисления отлично от $\top$ и $\bot$. Получившееся состояние памяти в таком случае обозначено через S'. Отсутствие на рис. 4.9 и 4.10 проверочного правила для текущей инструкции соответствует расширению S' текущими значениями P_p и P_B. Областями значений отображений P_p и P_B являются множества, включающие специальные значения $\bot$, а также непустые списки, первыми элементами которых являются переменные a_i, а последующими, необязательными элементами — смещения полей типа "массив структур" $o(f_s)$. Через $\|$ на рис. 4.9 обозначена конкатенация списков. Отношение $\blacktriangleright\!\!\blacktriangleright$ задает некоторое ограничение на исходные последовательности инструкций, которое, в частности, исключает существенное использование объединений и непрефиксных приведений типов указателей, а также вычитание указателей на элементы разных массивов. Тем не менее в заданное ограничение попадает достаточно существенный класс возможных последовательностей инструкций. В частности, ограниияения позволяют выделять память под произвольные структуры и адресовать все их поля до любого уровня вложенности. Сформулируем и приведем план доказательства следующей теоремы о полноте анализа регионов для соответствующего класса последовательностей инструкций:

Теорема 12 (Полнота анализа регионов)**.** *Для конечных последовательностей инструкций базового языка оо, для которых при любых значениях переменных n_{f_s}, любой достаточно большой разрядности адресов d ($d \geq d_{min}$ для некоторого d_{min}) и любых достаточно больших размерах используемой области памяти (задается константами l и u) выполнено оо $\blacktriangleright\!\!\blacktriangleright^* \top$, в контексте, для которого выполнено существование отображений τ, Δ, Δ^{-1} и Δ', удовлетворяющих соответствующим условиям* (11) *и* (12)*, выполнено*

$oo \rhd^* \top$.

Доказательство. Аналогично предыдущим теоремам, следствие $oo \blacktriangleright^* \top \implies oo \rhd^* \top$ доказывается по индукции через следствие предусловий. Из теорем 3 и 9 следует, что достаточно доказать следствие предусловий для правил deref', assign' и free' вывода отношения $\rhd$ из соответствующих предусловий правил вывода отношения $\blacktriangleright$ (включая все правила отношения $\blacktriangleright$). Проверка предусловия для правил deref', assign' и free' при выполненности соответствующих предусловий для правил вывода отношения $\blacktriangleright$ сводится к проверке выделенности некоторого множества адресов в некоторых соответствующих им регионах. Выделенность адресов в модельной семантике, то есть выполненность условий вида $M[(\rho, a, f_{s.\text{valid}})]$ доказывается от противного.

Вначале доказывается, что из ограничений (11) и (12) на контекст Γ и правил вывода отношения $\blacktriangleright$ следует инвариант $P_p[p_1] \neq \bot \wedge P_p[p_2] \neq \bot \implies P_p[p_1] = P_p[p_2] \implies \Gamma_\rho(p_1) = \Gamma_\rho(p_2)$ (достаточное условие равенства регионов), выполняющийся для всех состояний вычисления, отличных от $\top$ и $\bot$. Затем делается предположение, что в некотором состоянии вычисления выполнено предусловие одного из правил вывода deref, assign или free отношения $\blacktriangleright$, а также предусловие одного из соответствующих дополнительных правил для $\blacktriangleright$, но не выполнено соответствующее предусловие правила вывода отношения $\rhd$. Из вида предусловий рассматриваемых правил вывода и отсутствия предшествующих инструкций $\textbf{to\_int}(p, v)$ и $\textbf{of\_int}(p, s, v)$ (см. соответствующие правила для $\blacktriangleright$, дающие $\bot$) следует, что для выполненности предусловий в правилах для $\rhd$ достаточно, чтобы условия $M[(\rho, a, f_{s.\text{valid}})]$ проверялись для того же региона, для которого происходило соответствующее выделение памяти (назовем это достаточным условием выделенности адреса в регионе).

На каждом конкретном пути значение любого выделенного адреса памяти можно представить в виде формулы вида $a_i + \sum o(f_s) + c$, где c — константа, не зависящая от n_{f_s} (адреса выравнивающих полей из $\mathbb{F}_c$ отношением $\blacktriangleright\!\!\bigcirc$ делаются невыделенными). Теперь в силу невыполненности предусловия для правила вывода отношения $\rhd$ найдется такое указательное выражение p (без ограничения общности можно предполагать, что это переменная), что $M[(\Gamma_\rho(p), E_p[p], f_{\tau\left(\Gamma_\rho(p)\right).\mathrm{valid}})] \neq \widehat{1}$. Заметим теперь, что значение $P_p[p] \neq \bot$ (выполнено предусловие правила для $\blacktriangleright\!\!\bigcirc$) позволяет поставить p в соответствие формулу вида $a_i + \sum o(f_s) + v$, взяв a_i и $o(f_s)$ из $P_p[p]$ и $v = E_p[p] - a_i - \sum o(f_s)$. Из ограничений, задаваемых отношением $\blacktriangleright\!\!\bigcirc$ можно показать независимость значения v от a_i и $o(f_s)$. Разделим теперь все множество формул для выделенных адресов на подмножество тех, в которых префикс $a_i + \sum o(f_s)$ совпадает с соответствующим префиксом для значения p и тех, для которых он не совпадает. Теперь, объединив доказанный ранее инвариант о достаточном условии равенства регионов и достаточное условие выделенности адреса в регионе, получим что для всех формул из множества с совпадающим префиксом $c \neq v$.

Теперь во всех формулах с несовпадающим префиксом, а также в формуле для p выразим все значения $o(f_s)$ через n_{f_s} и некоторые константы. Запишем теперь условия неравенства значения формулы для p каждому из значений формул с несовпадающим префиксом. Заметим, что после приведения в неравенствах подобных слагаемых в каждом из них останется хотя бы по одной константе n_{f_s}. Так как v и все константы в полученных неравенствах не зависят от a_i и n_{f_s}, то найдутся такие значения этих переменных, что все полученные неравенства будут выполнены. В силу независимости значений переменных от a_i и n_{f_s}, а также произвольности d, l и u в условии теоремы, найдутся такие значения a_i и n_{f_s}, обеспечивающие полученные неравенства, при которых рассматриваемая инструкция будет также достижима в исход-

ной последовательности. При этом по-прежнему будут выполнены неравенства $c \neq v$. Это будет означать, что для переменной p не выполнено условие $V\big[E_p[p]\big]$ в исходной семантике, из чего следует $oo \blacktriangleright^* \bot$, что противоречит условию теоремы. □

4.5. Выводы

В данной главе описан новый вид анализа регионов для дедуктивной верификации Си-программ на основе базового языка с поддержкой произвольной вложенности структур, объединений и массивов в другие структуры. Определена исходная семантика базового языка, моделирование которой является задачей инструмента дедуктивной верификации. Предложены нормализующие преобразования, позволяющие преобразовать исходную аннотированную Си-программу, содержащую произвольные случаи использования объединений, префиксных и непрефиксных приведений типов указателей на структуры, а также конструкции для получения указателя на объемлющую структуру, в соответствующую программу на новом базовом языке. Представлена модельная семантика базового языка, задающая способ представления состояния памяти программ на базовом языке в виде логических формул, и доказаны теоремы о корректности и полноте этой семантики относительно исходной. Описаны дополнительные ограничения на контекст типизации термов базового языка, которые задают результат анализа регионов, обеспечивающего полное моделирование семантики входных программ, удовлетворяющих некоторому набору дополнительных ограничений. Приведена схема доказательства соответствующей теоремы о полноте анализа регионов.

Разработанная модели памяти позволяет расширить область применимости инструмента дедуктивной верификации Jessie на Си-программы, существенно использующие вложенные структуры и массивы, а также объедине-

ния и произвольные приведения типов указателей, без потери эффективности ранее реализованной в инструменте модели памяти с регионами. Разработанная модель памяти также предполагает более простой способ моделирования объединений и приведений типов указателей, не требующий введения с этой целью специального модельного состояния (используемой в исходном инструменте Jessie модельной таблицы тегов объектов). Моделирование этих возможностей языка Си интегрировано с моделированием защиты памяти и использует соответствующее модельное состояние (таблицу выделенности адресов V). Для разработанной модели памяти доказана корректность и полнота относительно формально заданной низкоуровневой семантики практически значимого (для индустриального кода) фрагмента языка Си, включающего поддержку защищенной динамической памяти, вложенных структур и массивов, а также объединений и приведений типов указателей. Предлагаемая модель памяти может быть достаточно легко расширена для поддержки структур и объединений с произвольным выравниванием полей, а также битовых полей структур и объединений. Частичная реализация предложенной модели памяти в инструменте Jessie – реализация поддержки вложенных структур и массивов, а также приведения типов указателей на целочисленные типы данных — позволила применить инструмент для дедуктивной верификации модуля ядра ОС Linux.

Предложенная в главе модель памяти с регионами так же, как и модель, представленная в статье [105], является нечувствительной к потоку управления, что ухудшает ее применимость для анализа рекурсивных структур данных. Функции, изменяющие данные, хранящихся внутри рекурсивных структур, могут использовать преимущества представления различных узлов рекурсивной структуры данных разными регионами. Однако в функциях, изменяющих сами структуры данных (то есть в функциях, выполняющих присваивания указателей на узлы этих структур), в большинстве случаев в силу

нечувствительности к потоку управления соответствующим указателям будет присвоен один и тот же регион. Совпадение регионов будет распространено также и на вызывающие функции. Поэтому в большинстве случаев предложенный анализ регионов не позволяет уменьшить число необходимых явных ограничений на непересечение адресов для рекурсивных структур данных. Однако он позволяет эффективно учесть эти ограничения при модификации данных, содержащихся в узлах этих структур, при условии, что соответствующий код, модифицирующий такие данные, представлен в виде отдельной функции.

Заключение

Основные научные и практические результаты, полученные в диссертационной работе и выносимые на защиту, состоят в следующем:

1. Разработана модель памяти на основе теории неинтерпретируемых функций для автоматической статической верификации Си-программ с использованием предикатных абстракций, уточняемых с помощью интерполяции Крейга;

2. Разработанная модель памяти на основе теории неинтерпретируемых функций интегрирована в инструмент автоматической статической верификации CPAchecker;

3. Разработана полная модель памяти с поддержкой вложенных структур и массивов, а также объединений и переинтерпретации типов указателей с автоматизированным разделенем на непересекающиеся области (регионы) для дедуктивной верификации Си-программ;

4. Предложена формализация низкоуровневой семантики практически значимого подмножества языка Си, а также соответствующие формальные доказательства корректности и полноты модели памяти с поддержкой вложенных структур и массивов для дедуктивной верификации Си-программ относительно этой семантики;

5. Предложена схема доказательства полноты метода автоматического разделения на непересекающиеся регионы при использовании соответствующей модели памяти для ограниченного класса Си-программ;

6. В инструменте дедуктивной верификации Jessie была реализована поддержка переинтерпретации типов указателей на целочисленные типы данных.

Развитие инструмента дедуктивной верификации Jessie ведется в отделе Технологий программирования Института системного программирования РАН, инструмент разработан командой ProVal в исследовательском центре INRIA Saclay — Île-de-France совместно с лабораторией LRI из исследовательского центра CNRS и университета Париж-Юг (Université Paris-Sud), расположенных в Орсе, Франция. Разработка инструмента автоматической статической верификации CPAchecker ведется в лаборатории "Software and Computational Systems Lab" университета Пассау (Германия) при участии отдела Технологий программирования ИСП РАН и непосредственном участии автора данной работы. Автор выражает свою признательность всем участникам соответствующих проектов. Большую помощью в выполнении работы оказал А.В.Хорошилов, а также В.С.Мутилин, Д.В. Ефремов, Dirk Beyer (университет Пассау) и Philipp Wendler (университет Пассау), которым автор выражает искреннюю благодарность.

Список литературы

1. Beyer D., Petrenko A. K. Linux Driver Verification // Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II, Ed. by T. Margaria, B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 1–6. ISBN: 978-3-642-34032-1. URL: http://dx.doi.org/10.1007/978-3-642-34032-1_1.
2. Mandrykin M. U., Khoroshilov A. V. Towards deductive verification of C programs with shared data // Programming and Computer Software. 2016. Vol. 42, no. 5. P. 324–332. URL: http://dx.doi.org/10.1134/S0361768816050054.
3. Mandrykin M. U., Khoroshilov A. V. Region analysis for deductive verification of C programs // Programming and Computer Software. 2016. Vol. 42, no. 5. P. 257–278. URL: http://dx.doi.org/10.1134/S0361768816050042.
4. Mandrykin M., Khoroshilov A. Towards deductive verification of concurrent Linux kernel code with Jessie // Computer Science and Information Technologies (CSIT), 2015. 2015. — Sept. P. 5–10.
5. Mandrykin M. U., Khoroshilov A. V. High-level memory model with low-level pointer cast support for Jessie intermediate language // Programming and Computer Software. 2015. Vol. 41, no. 4. P. 197–207. URL: http://dx.doi.org/10.1134/S0361768815040040.
6. Mandrykin M. U., Mutilin V. S., Petrenko A. K. et al. Configurable toolset for static verification of operating systems kernel modules // Programming and Computer Software. 2015. Vol. 41, no. 1. P. 49–64. URL: http://dx.doi.org/10.1134/S0361768815010065.

7. Löwe S., Mandrykin M., Wendler P. CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses // Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, Ed. by E. Ábrahám, K. Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. P. 392–394. ISBN: 978-3-642-54862-8. URL: http://dx.doi.org/10.1007/978-3-642-54862-8_27.
8. Mandrykin M. U., Mutilin V. S., Novikov E. M. et al. Using Linux Device Drivers for Static Verification Tools Benchmarking // Program. Comput. Softw. 2012. — sep. Vol. 38, no. 5. P. 245–256. URL: http://dx.doi.org/10.1134/S0361768812050039.
9. Shved P. E., Mutilin V. S., Mandrykin M. U. Experience of Improving the Blast Static Verification Tool // Program. Comput. Softw. 2012. — jun. Vol. 38, no. 3. P. 134–142. URL: http://dx.doi.org/10.1134/S0361768812030061.
10. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7 // Proceedings of TACAS. 2012. Vol. 7214. P. 525–527.
11. Мандрыкин М. У., Мутилин В. Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях // Труды Института системного программирования РАН. 2015. Vol. 27. P. 117–142.
12. Alpern B., Schneider F. B. Recognizing safety and liveness // Distributed Computing. 1987. Vol. 2, no. 3. P. 117–126. URL: http://dx.doi.org/10.1007/BF01782772.
13. Biere A., Artho C., Schuppan V. Liveness Checking as Safety Checking // Electronic Notes in Theoretical Computer Science. 2002. Vol. 66, no. 2. P. 160 – 177. URL: http://www.sciencedirect.com/science/article/pii/S1571066104804109.

14. Beyer D., Henzinger T. A., Jhala R., Majumdar R. The software model checker Blast: Applications to software engineering // Int. J. Softw. Tools Technol. Transf. 2007. Vol. 9, no. 5. P. 505–525.
15. Heizmann M., Hoenicke J., Podelski A. Software Model Checking for People Who Love Automata // Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Ed. by N. Sharygina, H. Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 36–52. ISBN: 978-3-642-39799-8. URL: http://dx.doi.org/10.1007/978-3-642-39799-8_2.
16. Clarke E., Kroening D., Yorav K. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking // Proceedings of the 40th Annual Design Automation Conference. DAC '03. New York, NY, USA: ACM, 2003. P. 368–371. URL: http://doi.acm.org/10.1145/775832.775928.
17. Clarke E., Kroening D., Yorav K. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking: Tech. rep.: Carnegie Mellon University, School of Computer Science, 2003.
18. Holzmann G. J. The Model Checker SPIN // IEEE Trans. Softw. Eng. 1997. — May. Vol. 23, no. 5. P. 279–295. URL: http://dx.doi.org/10.1109/32.588521.
19. Cimatti A., Clarke E. M., Giunchiglia E. et al. NuSMV 2: An OpenSource Tool for Symbolic Model Checking // Proceedings of the 14th International Conference on Computer Aided Verification. CAV '02. London, UK, UK: Springer-Verlag, 2002. P. 359–364. URL: http://dl.acm.org/citation.cfm?id=647771.734431.
20. Biere A., Cimatti A., Clarke E. M. et al. Bounded model checking // Advances in computers. 2003. Vol. 58. P. 117–148.
21. Clarke E., Biere A., Raimi R., Zhu Y. Bounded Model Checking Using Satisfiability Solving // Form. Methods Syst. Des. 2001. — Jul. Vol. 19, no. 1.

P. 7–34. URL: http://dx.doi.org/10.1023/A:1011276507260.

22. Biere A., Cimatti A., Clarke E. M., Zhu Y. Symbolic Model Checking Without BDDs // Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. TACAS '99. London, UK, UK: Springer-Verlag, 1999. P. 193–207. URL: http://dl.acm.org/citation.cfm?id=646483.691738.
23. Owre S., Rushby J. M., Shankar N. PVS: A Prototype Verification System // Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction. CADE-11. London, UK, UK: Springer-Verlag, 1992. P. 748–752. URL: http://dl.acm.org/citation.cfm?id=648230.752639.
24. Graf S., Saïdi H. Construction of Abstract State Graphs with PVS // Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel. 1997. P. 72–83.
25. Graf S., Saïdi H. Verifying invariants using theorem proving // Computer Aided Verification: 8th International Conference, CAV '96 New Brunswick, NJ, USA, July 31– August 3, 1996 Proceedings, Ed. by R. Alur, T. A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. P. 196–207. ISBN: 978-3-540-68599-9. URL: http://dx.doi.org/10.1007/3-540-61474-5_69.
26. Biere A. Bounded Model Checking // Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN: 1586039296, 9781586039295.
27. Amla N., Du X., Kuehlmann A. et al. An Analysis of SAT-based Model Checking Techniques in an Industrial Environment // Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods. CHARME'05. Berlin, Heidelberg: Springer-Verlag, 2005. P. 254–268. URL: http://dx.doi.org/10.1007/11560548_20.

28. Clarke E., Kroening D., Lerda F. A Tool for Checking ANSI-C Programs // Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings, Ed. by K. Jensen, A. Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 168–176. ISBN: 978-3-540-24730-2. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_15.
29. Cordeiro L., Fischer B., Marques-Silva J. SMT-Based Bounded Model Checking for Embedded ANSI-C Software // IEEE Transactions on Software Engineering. 2012. Vol. 38, no. 4. P. 957–974.
30. Merz F., Falke S., Sinz C. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR // Verified Software: Theories, Tools, Experiments: 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings, Ed. by R. Joshi, P. Müller, A. Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 146–161. ISBN: 978-3-642-27705-4. URL: http://dx.doi.org/10.1007/978-3-642-27705-4_12.
31. Ball T., Rajamani S. K. Bebop: A Symbolic Model Checker for Boolean Programs // Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. London, UK, UK: Springer-Verlag, 2000. P. 113–130. URL: http://dl.acm.org/citation.cfm?id=645880.672077.
32. Das S., Dill D. L. Successive approximation of abstract transition relations // Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on. 2001. P. 51–58.
33. Ball T., Cook B., Das S., Rajamani S. K. Refining Approximations in Software Predicate Abstraction // Tools and Algorithms for the Construc-

tion and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings, Ed. by K. Jensen, A. Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 388–403. ISBN: 978-3-540-24730-2. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_30.

34. Jhala R., McMillan K. L. Interpolant-based Transition Relation Approximation // Proceedings of the 17th International Conference on Computer Aided Verification. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005. P. 39–51. URL: http://dx.doi.org/10.1007/11513988_6.
35. Bradley A. R. SAT-based Model Checking Without Unrolling // Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'11. Berlin, Heidelberg: Springer-Verlag, 2011. P. 70–87. URL: http://dl.acm.org/citation.cfm?id=1946284.1946291.
36. Cimatti A., Griggio A. Software Model Checking via IC3 // Proceedings of the 24th International Conference on Computer Aided Verification. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012. P. 277–293. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_23.
37. Karpenkov E. G., Monniaux D., Wendler P. Program Analysis with Local Policy Iteration // Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings, Ed. by B. Jobstmann, M. K. R. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. P. 127–146. ISBN: 978-3-662-49122-5. URL: http://dx.doi.org/10.1007/978-3-662-49122-5_6.
38. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static driver verification with under 4% false alarms // Formal Methods in Computer-Aided Design

(FMCAD), 2010. 2010. — oct. P. 35 –42.

39. Clarke E., Kroening D., Sharygina N., Yorav K. Predicate Abstraction of ANSI-C Programs Using SAT // Form. Methods Syst. Des. 2004. — Sep. Vol. 25, no. 2-3. P. 105–127. URL: http://dx.doi.org/10.1023/B:FORM.0000040025.89719.f3.
40. Beyer D., Keremoglu M. E. CPAchecker: a tool for configurable software verification // Proceedings of the 23rd international conference on Computer aided verification. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011. P. 184–190.
41. Visser W., Havelund K., Brat G. et al. Model Checking Programs // Automated Software Engineering. 2003. Vol. 10, no. 2. P. 203–232. URL: http://dx.doi.org/10.1023/A:1022920129859.
42. Păsăreanu C. S., Visser W., Bushnell D. et al. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis // Automated Software Engineering. 2013. Vol. 20, no. 3. P. 391–425.
43. Klocwork Insight [Электронный ресурс] // Режим доступа: http://www.klocwork.com/products/insight/, свободный.
44. Coverity [Электронный ресурс] // Режим доступа: http://www.coverity.com/products/static-analysis.html, свободный.
45. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code // Programming and Computer Software. 2014. Vol. 40, no. 5. P. 265–275. URL: http://dx.doi.org/10.1134/S0361768814050041.
46. Nesov V. Automatically Finding Bugs in Open Source Programs // Third International Workshop on Foundations and Techniques for Open Source Software Certification. Vol. 20 of OpenCert 2009. 2009. P. 19–29.
47. Hoare C. A. R. An Axiomatic Basis for Computer Programming // Commun. ACM. 1969. — oct. Vol. 12, no. 10. P. 576–580. URL: http://doi.acm.

org/10.1145/363235.363259.

48. Floyd R. W. Assigning meaning to programs // Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia / Ed. by J. T. Schwartz. Vol. 19. Providence RI: American Mathematical Society, 1967. P. 19–31. [to get] first idea of termination function to prove termination of algorithms.
49. Dijkstra E. W. Guarded commands, nondeterminacy, and formal derivation of programs // Communications of the ACM. 1975. — August. Vol. 18, no. 8. P. 453–457.
50. Reynolds J. C. Separation Logic: A Logic for Shared Mutable Data Structures // Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002. P. 55–74. URL: http://dl.acm.org/citation.cfm?id=645683.664578.
51. Bertot Y., Castéran P., Huet G. (., Paulin-Mohring C. Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions. Texts in theoretical computer science. Berlin, New York: Springer, 2004. ISBN: 978-3-540-20854-9. Données complémentaires http://coq.inria.fr. URL: http://opac.inria.fr/record=b1101046.
52. Nipkow T., Wenzel M., Paulson L. C. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7.
53. Owre S., Rajan S., Rushby J. et al. PVS: Combining Specification, Proof Checking, and Model Checking // Computer-Aided Verification, CAV '96 / Ed. by R. Alur, T. A. Henzinger. Iss. Lecture Notes in Computer Science, no. 1102. New Brunswick, NJ: Springer-Verlag, 1996. — July/August. P. 411–414. URL: http://www.csl.sri.com/papers/pvs-cav96/.
54. Bobot F., Filliâtre J.-C., Marché C., Paskevich A. Why3: Shepherd Your

Herd of Provers // Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland: 2011. — August. URL: http://proval.lri.fr/submissions/boogie11.pdf.

55. Leino K. R. M., Wüstholz V. The Dafny Integrated Development Environment // Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. 2014. P. 3–15. URL: http://dx.doi.org/10.4204/EPTCS.149.2.
56. Leino K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness // Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010. P. 348–370. URL: http://dl.acm.org/citation.cfm?id=1939141.1939161.
57. Filliâtre J.-C., Paskevich A. Why3: Where Programs Meet Provers // Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Berlin, Heidelberg: Springer-Verlag, 2013. P. 125–128. URL: http://dx.doi.org/10.1007/978-3-642-37036-6_8.
58. Filliâtre J.-C., Marché C. Multi-prover Verification of C Programs // Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004. Proceedings, Ed. by J. Davies, W. Schulte, M. Barnett. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 15–29. ISBN: 978-3-540-30482-1. URL: http://dx.doi.org/10.1007/978-3-540-30482-1_10.
59. Moy Y. Automatic Modular Static Safety Checking for C Programs: Ph. D. thesis / Université Paris-Sud. 2009. — January. URL: http://www.lri.fr/~marche/moy09phd.pdf.
60. Cuoq P., Kirchner F., Kosmatov N. et al. Frama-C: A Software Analysis Perspective // Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Berlin, Heidelberg:

Springer-Verlag, 2012. P. 233–247. URL: http://dx.doi.org/10.1007/978-3-642-33826-7_16.

61. Delahaye M., Kosmatov N., Signoles J. Common Specification Language for Static and Dynamic Analysis of C Programs // Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC '13. New York, NY, USA: ACM, 2013. P. 1230–1235. URL: http://doi.acm.org/10.1145/2480362.2480593.
62. Ahrendt W., Beckert B., Hähnle R. et al. Verifying Object-oriented Programs with KeY: A Tutorial // Proceedings of the 5th International Conference on Formal Methods for Components and Objects. FMCO'06. Berlin, Heidelberg: Springer-Verlag, 2007. P. 70–101. URL: http://dl.acm.org/citation.cfm?id=1777707.1777713.
63. Burdy L., Cheon Y., Cok D. R. et al. An Overview of JML Tools and Applications // Int. J. Softw. Tools Technol. Transf. 2005. — jun. Vol. 7, no. 3. P. 212–232. URL: http://dx.doi.org/10.1007/s10009-004-0167-4.
64. Cohen E., Dahlweid M., Hillebrand M. et al. VCC: A Practical System for Verifying Concurrent C // Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009. P. 23–42. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_2.
65. Swamy N., Hriţcu C., Keller C. et al. Dependent Types and Multi-monadic Effects in F* // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '16. New York, NY, USA: ACM, 2016. P. 256–270. URL: http://doi.acm.org/10.1145/2837614.2837655.
66. Jacobs B., Smans J., Philippaerts P. et al. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java // Proceedings of the Third International Conference on NASA Formal Methods. NFM'11. Berlin, Heidelberg:

Springer-Verlag, 2011. P. 41–55. URL: http://dl.acm.org/citation.cfm?id=1986308.1986314.

67. Piskac R., Wies T., Zufferey D. GRASShopper - Complete Heap Verification with Mixed Specifications // Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. 2014. P. 124–139. URL: http://dx.doi.org/10.1007/978-3-642-54862-8_9.
68. Platzer A. An Object-oriented Dynamic Logic with Updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, 2004. — September. http://i12www.ira.uka.de/%7Ekey/doc/2004/odlMasterThesis.pdf.
69. Beckert B., Klebanov V. A Dynamic Logic for Deductive Verification of Concurrent Programs // Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK / Ed. by M. Hinchey, T. Margaria. IEEE Press, 2007.
70. Cohen E., Moskal M., Schulte W., Tobies S. A Practical Verification Methodology for Concurrent Programs: Tech. Rep. MSR-TR-2009-2019: Microsoft Research, 2009. — February. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=79554.
71. Norell U. Towards a practical programming language based on dependent type theory: Ph. D. thesis / Department of Computer Science and Engineering, Chalmers University of Technology. SE-412 96 Göteborg, Sweden, 2007. — September.
72. Brady E. C. IDRIS —: Systems Programming Meets Full Dependent Types // Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification. PLPV '11. New York, NY, USA: ACM, 2011. P. 43–54. URL: http://doi.acm.org/10.1145/1929529.1929536.

73. Fontaine P., Marion J.-Y., Merz S. et al. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants // Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'06. Berlin, Heidelberg: Springer-Verlag, 2006. P. 167–181. URL: http://dx.doi.org/10.1007/11691372_11.
74. Armand M., Faure G., Grégoire B. et al. A Modular Integration of SAT/SMT Solvers to Coq Through Proof Witnesses // Proceedings of the First International Conference on Certified Programs and Proofs. CPP'11. Berlin, Heidelberg: Springer-Verlag, 2011. P. 135–150. URL: http://dx.doi.org/10.1007/978-3-642-25379-9_12.
75. Henzinger T. A., Jhala R., Majumdar R., McMillan K. L. Abstractions from proofs // SIGPLAN Not. 2004. Vol. 39, no. 1. P. 232–244.
76. Кошелев В. К., Игнатьев В. Н., Борзилов А. И. Инфраструктура статического анализа программ на языке C# // Труды ИСП РАН. 2016. С. 21–40.
77. Pierce B. C. Untyped Systems // Types and Programming Languages. 1st edition. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.
78. Meyer B. Eiffel: The Language. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-247925-7.
79. Chen C., Xi H. Combining Programming with Theorem Proving // Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming. ICFP '05. New York, NY, USA: ACM, 2005. P. 66–77. URL: http://doi.acm.org/10.1145/1086365.1086375.
80. Developers T. R. P. The Rust Programming Language. 2016. URL: https://doc.rust-lang.org/book/ (дата обращения: 2016-08-24).
81. Мутилин В. Верификация драйверов операционной системы Linux при помощи предикатных абстракций: Ph. D. thesis / Институт

Системного Программирования РАН. 109004, г.Москва, ул. Александра Солженицына, 25, 2012. — ноябрь.

82. Cohen E., Dahlweid M., Hillebrand M. et al. VCC: A Practical System for Verifying Concurrent C // Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009. P. 23–42. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_2.
83. Barrett C., Deters M., de Moura L. et al. 6 Years of SMT-COMP // Journal of Automated Reasoning. 2013. Vol. 50, no. 3. P. 243–277. URL: http://dx.doi.org/10.1007/s10817-012-9246-5.
84. Cok D. R., Déharbe D., Weber T. The 2014 SMT Competition // Journal on Satisfiability, Boolean Modeling and Computation. 2014. Vol. 9. P. 207–242. URL: https://satassociation.org/jsat/index.php/jsat/article/view/122.
85. Conchon S., Déharbe D., Heizmann M., Weber T. SMT-COMP 2016. 2016. URL: http://smtcomp.sourceforge.net/2016/ (дата обращения: 2016-08-24).
86. Järvisalo M., Le Berre D., Roussel O., Simon L. The International SAT Solver Competitions // AI Magazine. 2012. Vol. 33, no. 1. P. 89–92.
87. Proceedings of SAT Competition 2016. Solver and Benchmark Descriptions / Ed. by T. Balyo, M. J. H. Heule, M. J. Järvisalo. Department of Computer Science Series of Publications B, University of Helsinki, Department of Computer Science, 2016.
88. Balyo T., Sinz C., Iser M., Biere A. SAT-Race 2015. 2015. URL: http://baldur.iti.kit.edu/sat-race-2015/ (дата обращения: 2016-08-24).
89. Condit J., Harren M., McPeak S. et al. CCured in the Real World // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI '03. New York, NY, USA: ACM, 2003.

P. 232–244. URL: http://doi.acm.org/10.1145/781131.781157.

90. Henzinger T. A., Jhala R., Majumdar R. Lazy abstraction // Symposium on Principles of Programming Languages. ACM Press, 2002. P. 58–70.
91. Shved P., Mutilin V., Mandrykin M. Static Verfication "Under The Hood": Implementation Details and Improvements of BLAST // Proceedings of SYRCoSE. Vol. 1. 2011. P. 54–60.
92. Andersen L. O. Program Analysis and Specialization for the C Programming Language. Københavns Universitet. Datalogisk Institut. DIKU, 1994.
93. Berndl M., Lhoták O., Qian F. et al. Points-to analysis using BDDs // SIGPLAN Not. 2003. Vol. 38, no. 5. P. 103–114.
94. Dijkstra E. W., Schölten C. S. The strongest postcondition // Predicate Calculus and Program Semantics. New York, NY: Springer New York, 1990. P. 209–215. ISBN: 978-1-4612-3228-5. URL: http://dx.doi.org/10.1007/978-1-4612-3228-5_12.
95. Kahn G. Natural Semantics // Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. STACS '87. London, UK, UK: Springer-Verlag, 1987. P. 22–39. URL: http://dl.acm.org/citation.cfm?id=646503.696269.
96. Khoroshilov A., Mutilin V., Novikov E. et al. Towards An Open Framework for C Verification Tools Benchmarking // Proceedings of PSI. 2011. P. 82–91.
97. Ball T., Bounimova E., Levin V., Moura L. D. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2: Tech. rep.: Microsoft Research, 2010. — March. URL: https://www.microsoft.com/en-us/research/publication/efficient-evaluation-of-pointer-predicates-with-z3-smt-solver-in-s
98. Ball T., Millstein T., Rajamani S. K. Polymorphic Predicate Abstraction // ACM Trans. Program. Lang. Syst. 2005. — mar. Vol. 27, no. 2. P. 314–343. URL: http://doi.acm.org/10.1145/1057387.1057391.

99. Ball T., Bounimova E., Cook B. et al. Thorough Static Analysis of Device Drivers // Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. EuroSys '06. New York, NY, USA: ACM, 2006. P. 73–85. URL: http://doi.acm.org/10.1145/1217935.1217943.
100. Cohen E., Moskal M., Tobies S., Schulte W. A Precise Yet Efficient Memory Model For C // Electron. Notes Theor. Comput. Sci. 2009. — October. Vol. 254. P. 85–103. URL: http://dx.doi.org/10.1016/j.entcs.2009.09.061.
101. Plotkin G. D. A structural approach to operational semantics. 1981.
102. Мандрыкин М. У., Хорошилов А. В. Анализ регионов для дедуктивной верификации Си-программ // Программирование. 2016. № 5. С. 3–29.
103. Burstall R. M. Some techniques for proving correctness of programs which alter data structures // Machine intelligence. 1972. Vol. 7, no. 23-50. P. 3.
104. Bornat R. Proving Pointer Programs in Hoare Logic // Proceedings of the 5th International Conference on Mathematics of Program Construction. MPC '00. London, UK, UK: Springer-Verlag, 2000. P. 102–126. URL: http://dl.acm.org/citation.cfm?id=648085.747307.
105. Hubert T., Marché C. Separation Analysis for Deductive Verification // Heap Analysis and Verification (HAV'07). Braga, Portugal: 2007. — March. P. 81–93. URL: http://www.lri.fr/~marche/hubert07hav.pdf.
106. Moy Y. Union and Cast in Deductive Verification // Proceedings of the C/C++ Verification Workshop. Vol. Technical Report ICIS-R07015. Radboud University Nijmegen, 2007. — July. P. 1–16. URL: http://www.lri.fr/~moy/Publis/moy07ccpp.pdf.
107. Мандрыкин М., Хорошилов А. Высокоуровневая модель памяти промежуточного языка Jessie с поддержкой произвольного приведения типов указателей // Программирование. 2015. Vol. 41. P. 23–39.
108. Böhme S., Moskal M. Heaps and Data Structures: A Challenge for Au-

tomated Provers // Proceedings of the 23rd International Conference on Automated Deduction. CADE'11. Berlin, Heidelberg: Springer-Verlag, 2011. P. 177–191. URL: http://dl.acm.org/citation.cfm?id=2032266.2032281.

109. Milner R. A theory of type polymorphism in programming // Journal of Computer and System Sciences. 1978. — 12. Vol. 17, no. 3. P. 348–375.
110. O'Hearn P. W., Reynolds J. C., Yang H. Local Reasoning About Programs That Alter Data Structures // Proceedings of the 15th International Workshop on Computer Science Logic. CSL '01. London, UK, UK: Springer-Verlag, 2001. P. 1–19. URL: http://dl.acm.org/citation.cfm?id=647851.737404.
111. Piskac R., Wies T., Zufferey D. Automating Separation Logic with Trees and Data // Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. New York, NY, USA: Springer-Verlag New York, Inc., 2014. P. 711–728. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_47.
112. Piskac R., Wies T., Zufferey D. Automating Separation Logic Using SMT // Proceedings of the 25th International Conference on Computer Aided Verification. CAV'13. Berlin, Heidelberg: Springer-Verlag, 2013. P. 773–789. URL: http://dx.doi.org/10.1007/978-3-642-39799-8_54.
113. Lewis H. R. Complexity Results for Classes of Quantificational Formulas // J. Comput. Syst. Sci. 1980. Vol. 21, no. 3. P. 317–353. URL: http://dx.doi.org/10.1016/0022-0000(80)90027-6.
114. Baudin P., Cuoq P., Filliâtre J.-C. et al. ACSL: ANSI/ISO C Specification Language.Version 1.11 — Aluminium-20160501, 2016. — September. URL: http://www.frama-c.com/download/frama-c-acsl-implementation.pdf.
115. Abadi A., Rabinovich A., Sagiv M. Decidable Fragments of Many-sorted

Logic // J. Symb. Comput. 2010. — feb. Vol. 45, no. 2. P. 153–172. URL: http://dx.doi.org/10.1016/j.jsc.2009.03.003.

116. Totla N., Wies T. Complete Instantiation-based Interpolation // Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '13. New York, NY, USA: ACM, 2013. P. 537–548. URL: http://doi.acm.org/10.1145/2429069.2429132.
117. Iguernelala M. Strengthening the Heart of an SMT-Solver: Design and Implementation of Efficient Decision Procedures: Thèse de doctorat / Université Paris-Sud. 2013. — jun.
118. TheThe Alt-Ergo SMT Solver. 2016. — September. URL: http://alt-ergo.lri.fr/.
119. Barrett C., Tinelli C. CVC3 // Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings, Ed. by W. Damm, H. Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. P. 298–302. ISBN: 978-3-540-73368-3. URL: http://dx.doi.org/10.1007/978-3-540-73368-3_34.
120. Barrett C., Conway C. L., Deters M. et al. CVC4 // Proceedings of the 23rd International Conference on Computer Aided Verification. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011. P. 171–177. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.
121. De Moura L., Bjørner N. Z3: An Efficient SMT Solver // Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008. P. 337–340. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.
122. Мандрыкин М. У., Мутилин В. С., Новиков Е. М. и др. Использование драйверов устройств операционной системы Linux для сравнения

инструментов статической верификации // Программирование. 2012. Т. 5. С. 54–71.

123. Muller P., Vojnar T. CPAlien: Shape Analyzer for CPAChecker // Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, Ed. by E. Ábrahám, K. Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. P. 395–397. ISBN: 978-3-642-54862-8. URL: http://dx.doi.org/10.1007/978-3-642-54862-8_28.
124. Beyer D. Competition on Software Verification // Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings, Ed. by C. Flanagan, B. König. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 504–524. ISBN: 978-3-642-28756-5. URL: http://dx.doi.org/10.1007/978-3-642-28756-5_38.
125. Beyer D. Second Competition on Software Verification // Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Ed. by N. Piterman, S. A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 594–609. ISBN: 978-3-642-36742-7. URL: http://dx.doi.org/10.1007/978-3-642-36742-7_43.
126. Beyer D. Competition on Software Verification (SV-COMP). 2016. — September. URL: http://sv-comp.sosy-lab.org/.
127. Ball T., Levin V., Rajamani S. K. A decade of software model checking with SLAM // Commun. ACM. 2011. Vol. 54, no. 7. P. 68–76.
128. Ball T., Cook B., Levin V., Rajamani S. K. SLAM and Static Driver Veri-

fier: technology transfer of formal methods inside Microsoft: Tech. rep.: Microsoft Research, 2004. URL: ftp://ftp.research.microsoft.com/pub/tr/tr-2004-08.pdf.

129. Ball T., Cook B., Levin V., Rajamani S. K. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft // Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings, Ed. by E. A. Boiten, J. Derrick, G. Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 1–20. ISBN: 978-3-540-24756-2. URL: http://dx.doi.org/10.1007/978-3-540-24756-2_1.
130. Craig W. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory // Journal of Symbolic Logic. 1957. — Sep. Vol. 22, no. 3. P. 269–285. URL: https://www.cambridge.org/core/article/three-uses-of-the-herbrand-gentzen-theorem-in-relating-model-theor 7674DE501824D8FC294FB396CD5617DB.
131. Lyndon R. C. An interpolation theorem in the predicate calculus. // Pacific J. Math. 1959. Vol. 9, no. 1. P. 129–142. URL: http://projecteuclid.org/euclid.pjm/1103039458.
132. Web-site. Linux Deductive Verification. http://linuxtesting.ru/astraver.
133. Beyer D., Keremoglu M., Wendler P. Predicate Abstraction with Adjustable-Block Encoding // Formal Methods in Computer-Aided Design, 2010. FMCAD 2010. 2010.
134. Ball T., Podelski A., Rajamani S. K. Boolean and Cartesian Abstraction for Model Checking C Programs // Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings, Ed. by T. Margaria, W. Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

P. 268–283. ISBN: 978-3-540-45319-2. URL: http://dx.doi.org/10.1007/3-540-45319-9_19.

135. Lahiri S. K., Nieuwenhuis R., Oliveras A. SMT Techniques for Fast Predicate Abstraction // Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings, Ed. by T. Ball, R. B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 424–437. ISBN: 978-3-540-37411-4. URL: http://dx.doi.org/10.1007/11817963_39.
136. Мандрыкин М., Мутилин В., Хорошилов А. Введение в метод CEGAR – уточнение абстракции по контрпримерам // Труды Института системного программирования РАН. 2013. Vol. 24. P. 37.
137. Beyer D., Henzinger T. A., Théoduloz G. Configurable software verification: concretizing the convergence of model checking and program analysis // Proceedings of CAV. Berlin, Heidelberg: Springer-Verlag, 2007. P. 504–518.
138. Beyer D. Competition on Software Verification (SV-COMP). 2016. — September. URL: http://sv-comp.sosy-lab.org/2016.
139. Захаров И. С., Мандрыкин М. У., Mutilin V. S. и др. Конфигурируемая система статической верификации модулей ядра операционных систем // Программирование. 2015. Т. 41, № 1. С. 44–67. Ядро операционной системы (ОС) представляет собой критичную в отношении надежности и производительности программную систему. Качество ядра современных ОС уже находится на достаточно высоком уровне. Иначе обстоит дело с модулями ядра, например, драйверами устройств, которые по ряду причин имеют существенно более низкий уровень качества. Одними из наиболее критичных и распространенных ошибок в модулях ядра являются нарушения правил корректного использования программного интерфейса ядра ОС. Выявить все такие нарушения в модулях или доказать их корректность потенциально можно с помощью инструментов

статической верификации, которым для проведения анализа необходимо предоставить контрактные спецификации, описывающие формальным образом обязательства ядра и модулей по отношению друг к другу. В статье рассматриваются существующие методы и системы статической верификации модулей ядра различных ОС. Предлагается новый метод статической верификации модулей ядра ОС Linux, который позволяет конфигурировать процесс проверки на каждом из его этапов. Показывается, каким образом данный метод может быть адаптирован для проверки компонентов ядра других ОС. Описывается архитектура конфигурируемой системы статической верификации модулей ядра ОС Linux, реализующей предложенный метод, и демонстрируются результаты ее практического применения. В заключении рассматриваются направления дальнейшего развития.

140. Lukasczyk S. Unbounded Heap Support for CPAchecker's Predicate Analysis Using SMT Arrays. 2016.
141. Mcmillan K. L. Lazy abstraction with interpolants // In Proc. CAV, LNCS 4144. Springer, 2006. P. 123–136.
142. Dijkstra E. A Discipline of Programming. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN: 9780132158718. URL: https://books.google.ru/books?id=MsUmAAAAMAAJ.
143. Kroening D., Strichman O. Decision Procedures: An Algorithmic Point of View. 1 edition. Springer Publishing Company, Incorporated, 2008. ISBN: 3540741046, 9783540741046.
144. Cook S. A. The Complexity of Theorem-proving Procedures // Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71. New York, NY, USA: ACM, 1971. P. 151–158. URL: http://doi.acm.org/10.1145/800157.805047.
145. Davis M., Logemann G., Loveland D. A Machine Program for Theorem-prov-

ing // Commun. ACM. 1962. — jul. Vol. 5, no. 7. P. 394–397. URL: http://doi.acm.org/10.1145/368273.368557.

146. Kroening D., Strichman O. Decision Procedures for Propositional Logic // Decision Procedures: An Algorithmic Point of View. 1 edition. Springer Publishing Company, Incorporated, 2008. ISBN: 3540741046, 9783540741046.
147. Robinson J. A. A Machine-Oriented Logic Based on the Resolution Principle // J. ACM. 1965. — jan. Vol. 12, no. 1. P. 23–41. URL: http://doi.acm.org/10.1145/321250.321253.
148. Gomes C. P., Kautz H., Sabharwal A., Selman B. Satisfiability Solvers // Handbook of Knowledge Representation. San Diego, USA: Elsevier Science, 2007. ISBN: 0444522115, 9780444522115.
149. Mitchell D., Selman B., Levesque H. Hard and Easy Distributions of SAT Problems // Proceedings of the Tenth National Conference on Artificial Intelligence. AAAI'92. AAAI Press, 1992. P. 459–465. URL: http://dl.acm.org/citation.cfm?id=1867135.1867206.
150. Bryant R. E., Velev M. N. Boolean Satisfiability with Transitivity Constraints // ACM Trans. Comput. Logic. 2002. — oct. Vol. 3, no. 4. P. 604–627. URL: http://doi.acm.org/10.1145/566385.566390.
151. Nelson G., Oppen D. C. Fast Decision Procedures Based on Congruence Closure // J. ACM. 1980. — apr. Vol. 27, no. 2. P. 356–364. URL: http://doi.acm.org/10.1145/322186.322198.
152. Shostak R. E. An Algorithm for Reasoning About Equality // Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 1. IJCAI'77. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1977. P. 526–527. URL: http://dl.acm.org/citation.cfm?id=1624435.1624551.
153. Хачиян Л. Г. Полиномиальные алгоритмы в линейном программировании // Журнал вычислительной математики и

математической физики. 1980. Vol. 20. P. 51–68.

154. Khachiyan L. Polynomial algorithms in linear programming // USSR Computational Mathematics and Mathematical Physics. 1980. Vol. 20, no. 1. P. 53 – 72. URL: http://www.sciencedirect.com/science/article/pii/0041555380900610.

155. Papadimitriou C. H. On the Complexity of Integer Programming // J. ACM. 1981. — oct. Vol. 28, no. 4. P. 765–768. URL: http://doi.acm.org/10.1145/322276.322287.

156. Oppen D. C. Complexity, convexity and combinations of theories // Theoretical Computer Science. 1980. Vol. 12, no. 3. P. 291 – 302. URL: http://www.sciencedirect.com/science/article/pii/0304397580900596.

157. Tinelli C., Harandi M. A New Correctness Proof of the Nelson-Oppen Combination Procedure // Frontiers of Combining Systems: First International Workshop, Munich, March 1996, Ed. by F. Baader, K. U. Schulz. Dordrecht: Springer Netherlands, 1996. P. 103–119. ISBN: 978-94-009-0349-4. URL: http://dx.doi.org/10.1007/978-94-009-0349-4_5.

158. Tarski A. A Decision Method for Elementary Algebra and Geometry // Quantifier Elimination and Cylindrical Algebraic Decomposition, Ed. by B. F. Caviness, J. R. Johnson. Vienna: Springer Vienna, 1998. P. 24–84. ISBN: 978-3-7091-9459-1. URL: http://dx.doi.org/10.1007/978-3-7091-9459-1_3.

159. Tarski A. A Decision Method for Elementary Algebra and Geometry. University of California Press Berkeley and Los Angeles, 1948.

160. Collins G. E. Quantifier elimination for real closed fields by cylindrical algebraic decompostion // Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975, Ed. by H. Brakhage. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975. P. 134–183. ISBN: 978-3-540-37923-2. URL: http://dx.doi.org/10.1007/

3-540-07407-4_17.

161. Kapur D. Using Gröbner Bases to Reason About Geometry Problems // J. Symb. Comput. 1986. — dec. Vol. 2, no. 4. P. 399–408. URL: http://dx.doi.org/10.1016/S0747-7171(86)80007-4.
162. Fränzle M., Herde C., Teige T. et al. Efficient Solving of Large Non-Linear Arithmetic Constraint Systems with Complex Boolean Structure // Journal on Satisfiability, Boolean Modeling, and Computation. 2007. Vol. 1.
163. Gödel K. Über Formal Unentscheidbare Sätze der Principia Mathematica Und Verwandter Systeme I // Monatshefte für Mathematik. 1931. Vol. 38, no. 1. P. 173–198.
164. Conchon S., Iguernelala M., Mebsout A. A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo // Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. SYNASC '13. Washington, DC, USA: IEEE Computer Society, 2013. P. 161–168. URL: http://dx.doi.org/10.1109/SYNASC.2013.29.
165. Bozzano M., Bruttomesso R., Cimatti A. et al. Encoding RTL Constructs for MathSAT: A Preliminary Report // Electron. Notes Theor. Comput. Sci. 2006. — jan. Vol. 144, no. 2. P. 3–14. URL: http://dx.doi.org/10.1016/j.entcs.2005.12.001.
166. Ganai M. K. Efficient Decision Procedure for Bounded Integer Non-linear Operations Using SMT($\mathcal{LIA}$) // Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing. HVC '08. Berlin, Heidelberg: Springer-Verlag, 2009. P. 68–83. URL: http://dx.doi.org/10.1007/978-3-642-01702-5_11.
167. Stump A., Barrett C. W., Dill D. L., Levitt J. A Decision Procedure for an Extensional Theory of Arrays // Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS '01). IEEE Computer Society, 2001. —

jun. P. 29–37. Boston, Massachusetts. URL: http://www.cs.nyu.edu/~barrett/pubs/SBDL01.pdf.

168. Bradley A. R., Manna Z., Sipma H. B. What's Decidable About Arrays? // Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings, Ed. by E. A. Emerson, K. S. Namjoshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 427–442. ISBN: 978-3-540-31622-0. URL: http://dx.doi.org/10.1007/11609773_28.
169. Papadimitriou C. H. Computational Complexity // Encyclopedia of Computer Science. Chichester, UK: John Wiley and Sons Ltd. P. 260–265. URL: http://dl.acm.org/citation.cfm?id=1074100.1074233.
170. Church A. An Unsolvable Problem of Elementary Number Theory // American Journal of Mathematics. 1936. — apr. Vol. 58, no. 2. P. 345–363. URL: http://dx.doi.org/10.2307/2371045.
171. Turing A. M. On Computable Numbers, with an Application to the Entscheidungsproblem // Proceedings of the London Mathematical Society. 1936. Vol. 2, no. 42. P. 230–265. URL: http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf. Turing's famous demonstration of the formal limits on computation based on a proof that the *halting problem* is undecidable.
172. Sutcliffe G. The 6th IJCAR Automated Theorem Proving System Competition –CASC-J6 // AI Commun. 2013. — apr. Vol. 26, no. 2. P. 211–223. URL: http://dl.acm.org/citation.cfm?id=2594582.2594586.
173. Sutcliffe G., Suttner C. The TPTP Problem Library for Automated Theorem Proving. 2016. — September. URL: http://www.cs.miami.edu/~tptp/.
174. Fischer M. J., Rabin M. O. SUPER-EXPONENTIAL COMPLEXITY OF PRESBURGER ARITHMETIC: Tech. rep. Cambridge, MA, USA: 1974.

Свидетельства о государственной регистрации программ для ЭВМ

1. Мандрыкин М.У., Хорошилов А.В. “Программа дедуктивной верификации программ на языке программирования Си, использующая модель памяти с чувствительным к контексту разделением на непересекающиеся регионы”. Свидетельство о государственной регистрации программы для ЭВМ №2016618347 от 27.07.2016.

2. Мандрыкин М.У., Хорошилов А.В. “Система дедуктивной верификации программ с возможностью адаптивного моделирования операций над целочисленными данными”. Свидетельство о государственной регистрации программы для ЭВМ №2016618348 от 27.07.2016.

3. Мандрыкин М.У., Хорошилов А.В. “Программа дедуктивной верификации программ на языке Си с возможностью интерпретации участков памяти как объектов разных типов”. Свидетельство о государственной регистрации программы для ЭВМ №2015617941 от 27.07.2015.

4. Мандрыкин М.У., Мутилин В.С., Хорошилов А.В. “Построитель формул с моделированием памяти для уточнения предикатной абстракции с помощью интерполяции”. Свидетельство о государственной регистрации программы для ЭВМ №2013614375 от 06.05.2013.

5. Швед П.Е., Новиков Е.М., Мандрыкин М.У., Мутилин В.С., Хорошилов А.В. “Система проверки выполнения проблемно-ориентированных правил для Си программ”. Свидетельство о государственной регистрации программы для ЭВМ №2012615596 от 20.06.2012.

6. Мандрыкин М.У., Мутилин В.С., Хорошилов А.В. “Интерполирующий

решатель, поддерживающий формулы с кванторами в теории линейной арифметики и неинтерпретируемых функций". Свидетельство о государственной регистрации программы для ЭВМ №2012618566 от 21.09.2012.

Приложение А

Теории, поддерживаемые современными решателями

Рассмотрим основные логические теории, доступные в большинстве современных решателей. Для каждой рассматриваемой теории опишем грамматику соответствующих формул, аксиоматизацию теории, в случаях, где она не является широко известной, а также укажем алгоритмическую разрешимость или сложность соответствующих решающих процедур. Также укажем алгоритмическую разрешимость наиболее часто используемых комбинаций логических теорий и приведем некоторые характеристики формул в теориях, использованных на практике при решении индустриальных задач, на основе тестовых наборов из индексов SMT-COMP'16 [83, 84, 85] и SAT-Race'15 [86, 87, 88]. Характеристики формул приведены в таблице А.1. В этой таблице приведены данные о количестве формул в тестовых наборах (второй столбец), среднем и максимальном количестве переменных (неинтерпретируемых констант и функций) в формулах для каждого набора (столбцы 3 –18), а также среднем и максимальном размерах тестовых формул (два последних столбца). Используемые в формулах теории и их комбинации обозначены так же, как в SMT-COMP, расшифровка соответствующих сокращений также приведена в таблице и далее в описаниях теорий. В таблице приведены данные только для формул, успешно разрешенных хотя бы одним из решателей в рамках соответствующих ограничений по времени выполнения и памяти. Приведены данные только для категорий, содержащих не менее 500 таких успешно разрешенных тестовых формул. Для категории SAT использовался лимит времени 5000 секунд (около 83 минут) и памяти 24ГБ [88], для остальных категорий — 2400 секунд и 60ГБ [85].

Логика	К-во ф-л	Количество переменных по типам																Размер ф-лы, КБ	
		Bool		Int		Real		_ BitVec n		Array i e		FloatingPoint e s		не Bool		функции			
		срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.	срд.	мкс.
SAT	651	83 119	1 614 656															13 057	166 890
QF_UF	6 649	0	0											7	300	1	44	111	6 036
QF_LRA	1 626	135	1 986			2 327	304 937							2 327	304 937			769	57 570
QF_LIA	5 839	1310	7 992	495	87 482									495	87 482			262	27 144
QF_IDL	2 094	14 875	467 104	6 577	115 225									6 577	115 255			1 409	48 727
QF_NRA	10 245	59	128			260	67 645							260	67 645			55	9 802
QF_NIA	8 593	11	464	19	2 606									19	2 606			47	39 089
QF_BV	32 476	21 779	828 508					711	635 313					711	635 313			618	390 770
QF_FP	40 119	0	0									2	6	2	6	1	1	1	2
QF_AX	551	0	0							20	121			55	241			11	62
QF_UFLRA	1627	33	50			146	14 634							146	14 634	16	3 149	782	161 851
QF_UFLIA	598	0	0	9	13									9	13	5	20	11	71
QF_AUFLIA	1009	20	43	36	123					27	121			64	244	1	19	8	145
QF_ABV	14 917	9 579	85 755					798	1 528 293	8	1 713			798	1 528 293			156	82 279
UF	2 839													54	991	78	2 833	38	327
NRA	3 788															0	0	2	3
UFLIA	8 404															63	1 166	50	488
UFNIA	2 319															165	669	350	4 153
AUFLIRA	19 849															20	100	10	193
AUFNIRA	1 050															18	195	11	94

Таблица А.1. Характеристики тестовых формул из тестовых наборов SMT-COMP'16 и SAT-Race'15. Обозначения теорий: UF — неинтерпретируемые функции, LRA — линейная вещественная арифметика, LIA — линейная целочисленная арифметика, IDL — теория целочисленных разностей, NRA — нелинейная вещественная арифметика, NIA — нелинейная целочисленная арифметика, BV — битовые векторы конечной длины, FP — операции над числами с плавающей точкой, A или AX — массивы, QF — отсутствие кванторов.

A.1. Пропозициональная логика

Пропозициональная логика является широко известной, минимальная грамматика соответствующих логических формул выглядит следующим образом:

$$
\begin{array}{lll}
\textit{формула} & ::= & \textit{формула} \wedge \textit{формула} \\
 & | & \neg\ \textit{формула} \\
 & | & (\ \textit{формула}\ ) \\
 & | & \textit{атом} \\
\textit{атом} & ::= & \textit{предикат} \mid \top \mid \bot \\
\textit{предикат} & ::= & \textit{неинтерпретируемая\_логическая\_константа}
\end{array}
$$

$\wedge$ обозначает дизъюнкцию, $\neg$ — отрицание, $\top$ и $\bot$ — логические константы “истина” и “ложь” соответственно. Другие логические связки, такие как $\vee$ (дизъюнкция) и $\rightarrow$ (импликация) могут быть выражены с использованием связок $\wedge$ и $\neg$. Предикаты в пропозициональной логике не имеют аргументов, не зависят от какого-либо неявного состояния и не наделяются какой-либо интерпретиацией, то есть являются неинтерпретируемыми логическими константами.

Задача выполнимости формул в пропозициональной логике является NP-полной [144], используемые в решателях оптимизированные алгоритмы на основе общих алгоритмов DPLL [145, 146] и резолютивного вывода [147] в худшем случае работают за время, экспоненциально зависящее от числа предикатов, однако на практике в зависимости от вида входной формулы могут проявлять как экспоненциальный, так и квазилинейный рост времени работы, решая за время порядка 90 минут задачи с количеством предикатов от нескольких десятков до нескольких миллионов [148]. Следует отметить, что квазилинейный рост времени работы характерен для статистического боль-

шинства задач [149]. Как видно из таблицы A.1, современные инструменты способны разрешать формулы с количеством переменных порядка ста тысяч, причем максимальное число переменных доходит до полутора миллионов.

Если считать исходную формулу заданной в конъюнктивной нормальной форме, то решатели формул в пропозициональной логике используют объем памяти, зависимость которого от числа предикатов в формуле близка к линейной.

A.2. Теория равенства

Теория равенства расширяет пропозициональную логику одним символом отношения — предикатом равенства =. Соответствующее расширение грамматики логических формул выглядит следующим образом:

$$
\begin{array}{lll}
\textit{предикат} & ::= & \ldots \\
 & | & \textit{терм} = \textit{терм} \\
\textit{терм} & ::= & \textit{неинтерпретируемая\_константа}
\end{array}
$$

Задача выполнимости формул в пропозициональной логике, расширенной теорией равенства, также является NP-полной [143]; как и для формул без равенства, время разрешения формул с равенством на практике может сильно различаться, но для индустриальных задач часто квазилинейно зависит от числа неинтерпретируемых констант и предикатов. Зависимость объема используемой памяти от числа неинтерпретируемых констант и предикатов близка к линейной. На практике теория равенства как правило используется совместно с теорией неинтерпретируемых функций и кванторами, поэтому в тестовых наборах SMT-COMP отсутствует категория для формул в пропозициональной логике с равенством без неинтерпретируемых функций или кванторов.

A.3. Теория неинтерпретируемых функций

Теория неинтерпретируемых функций расширяет теорию равенства введением произвольного набора неинтерпретируемых функциональных и предикатных символов и соответствующих им термов и предикатов:

предикат ::= ...
| *неинт._пред._символ* (*непустой_список_термов*)

терм ::= ...
| *неинт._функц._символ* (*непустой_список_термов*)

непустой_список_термов ::= *терм список_термов*

список_термов ::= ϵ | , *терм список_термов*

Для расширения пропозициональной логики с равенством неинтерпретируемыми функциями и предикатами достаточно введения одной аксиомы для каждого функционального или предикатного символа. Соответствующие аксиомы называются аксиомами *функциональной консистентности* или, иначе, аксиомами *конгруэнтного замыкания* (англ. *congruence closure*) и имеют вид:

$$\forall t_1, \ldots t_n, t'_1, \ldots, t'_n . \bigwedge_{i=n}^{n} t_i = t'_i \implies f(t_1, \ldots, t_n) = f(t'_1, \ldots, t_n)$$

для функций и

$$\forall t_1, \ldots t_n, t'_1, \ldots, t'_n . \bigwedge_{i=n}^{n} t_i = t'_i \implies \big(P(t_1, \ldots, t_n) \iff P(t'_1, \ldots, t_n)\big)$$

для предикатов.

Если за число неинтерпретируемых констант и предикатов взять число всех различных термов и предикатов, входящих в исходную формулу, то задача выполнимости формул в пропозициональной логике с равенством и

неинтерпретируемыми функциями обладает примерно теми же характеристиками, что и задача разрешимости формул в логике с равенством, так как может быть решена с помощью полиномиальной по времени и памяти редукции к этой задаче [150]. Алгоритмы разрешения задач выполнимости для формул с равенством и неинтерпретируемыми функциями описаны, например, в [151, 152]. В таблице A.1 пропозициональная логика с равенством и неинтерпретируемыми функциями обозначена как QF_UF.

A.4. Теории линейной вещественной и целочисленной арифметики

Теории линейной вещественной и целочисленной арифметики расширяют пропозициональную логику конъюнкциями линейных ограничений — неравенств. Соответствующее расширение пропозициональной логики:

$$
\begin{aligned}
\textit{предикат} &::= \ldots \\
&\quad \mid \textit{сумма отношение сумма} \\
\textit{отношение} &::= \leq \mid < \\
\textit{сумма} &::= \textit{терм} \mid \textit{сумма} + \textit{терм} \\
\textit{терм} &::= \textit{неинт.\_числовая\_константа} \\
&\quad \mid \textit{числовая\_константа} \\
&\quad \mid \textit{числовая\_константа} \times \textit{неинт.\_числовая\_константа}
\end{aligned}
$$

Доменами числовых констант (как интерпретируемых, то есть собственно чисел, так и неинтерпретируемых, то есть переменных в математическом смысле) могут быть множества вещественных и целых чисел. Алгоритмы разрешения формул для теории линейной арифметики непосредственно работают только с конъюнкциями линейных неравенств. Задача разрешения

конъюнкции линейных неравенств для домена вещественных чисел является полиномиальной от битового размера входной формулы (слабо полиномиальной) [153, 154], а для домена целых чисел — NP-полной [155]. Задача выполнимости формул общего вида в этой теории решается с помощью комбинирования решающей процедуры для конъюнкции линейных неравенств и основной решающей процедуры для формул пропозициональной логики с равенством. Используемые для комбинирования алгоритмы описаны, например, в [156, 157]. Решатели для обеих теорий широко используются для решения индустриальных задач. В таблице A.1 пропозициональная логика с вещественной линейной арифметикой обозначена как QF_LRA, а с линейной целочисленной арифметикой — как QF_LIA.

Широко известным фрагментом теории линейной арифметики является теория разностей, в которой все неравенства имеют вид $x+y < c$ или $x+y \leq c$, где c — числовая константа. Задача о выполнимости конъюнкции разностей являются полиномиальной как для домена вещественных, так и для домена целых чисел. Однако данный фрагмент в силу слабой выразительности не представляет большого интереса в контексте верификации промышленных программ на языке Си. В таблице A.1 приведены данные только для теории целочисленных разностей, соответствующая логика обозначена как QF_IDL.

A.5. Теории нелинейной вещественной и целочисленной арифметики

Теория нелинейной вещественной арифметики расширяет пропозициональную логику конъюнкциями произвольных неравенств термов, использующих операции сложения, умножения и деления:

предикат ::= ...
| *терм отношение терм*

$$
\begin{aligned}
&\textit{отношение} && ::= \leq \mid < \\
&\textit{терм} && ::= \textit{неинт.\_числовая\_константа} \\
&&& \mid \textit{числовая\_константа} \\
&&& \mid \textit{терм} + \textit{терм} \\
&&& \mid \textit{терм} \times \textit{терм} \\
&&& \mid \textit{терм} \div \textit{терм}
\end{aligned}
$$

Известно, что задача о выполнимости формул в пропозициональной логике с нелинейной вещественной арифметикой является алгоритмически разрешимой [158, 159], для её решения существуют и разрабатываются различные алгоритмы [160, 161, 162], как правило, имеющие в худшем случае сложность по времени не более двойной экспоненты от числа неинтерпретируемых числовых констант (что соответствует сложности широко известного алгоритма цилиндрической алгебраической декомпозиции [160]). Как видно из таблицы A.1, где соответствующая логика обозначена как QF_NRA, в индустрии автоматические решатели применяются для формул в теории нелинейной вещественно арифметики с количеством неинтерпретируемых констант порядка нескольких сотен, в некоторых успешно разрешаемых формулах число переменных доходит до нескольких десятков тысяч.

Теория нелинейной целочисленной арифметики аналогично добавляет неравенства термов, использующих операции сложения, умножения, деления нацело, получения остатка от деления и взятия модуля (абсолютного значения) целого числа:

$$
\begin{aligned}
&\textit{предикат} && ::= \ldots \\
&&& \mid \textit{терм отношение терм} \\
&\textit{отношение} && ::= \leq \mid <
\end{aligned}
$$

$$
\begin{aligned}
\textit{терм} \quad ::= \; & \textit{неинт.\_числовая\_константа} \\
\mid \; & \textit{числовая\_константа} \\
\mid \; & |\textit{терм}| \\
\mid \; & \textit{терм} + \textit{терм} \\
\mid \; & \textit{терм} \times \textit{терм} \\
\mid \; & \textit{терм} \div \textit{терм} \\
\mid \; & \textit{терм} \bmod \textit{терм}
\end{aligned}
$$

Соответствующая задача выполнимости логических формул алгоритмически неразрешима [163]. Тем не менее, как видно из таблицы A.1 (логика QF_NIA), используемые на практике эвристические (неполные) алгоритмы (например, [164, 165, 166]) позволяют разрешать некоторые формулы в пропозициональной логике с нелинейной целочисленной арифметикой, содержащие в среднем несколько десятков целочисленных переменных.

A.6. Теория битовых векторов конечной длины

Теория битовых векторов расширяет пропозициональную логику разнообразными символами отношений и функций, объединенных интерпретацией их аргументов как массивов конечной фиксированной наперед заданной длины, элементами которых являются логические значения. Набор включаемых в теорию символов функций и предикатов довольно велик, так как соответствует всевозможным операциям над битовыми представлениями, поддерживаемым современными процессорами, включая арифметические операции над знаковыми и беззнаковыми целыми, операции знакового и беззнакового расширения, операции сравнения, побитовые логические операции, сдвиги, повороты, операции выделения диапазона битов и конкатенации битовых векторов. Приведем грамматику расширения пропозициональной логики сим-

волами функций и предикатов над битовыми векторами конечной длины, которые будут использоваться далее в данной работе:

$$
\begin{array}{lll}
\textit{предикат} & ::= & \ldots \\
 & | & \textit{терм отношение терм} \\
\textit{отношение} & ::= & \widehat{<}_S \mid \widehat{<}_U \mid = \\
\textit{терм} & ::= & \textit{терм операция терм} \\
 & | & \textit{неинт.\_константный\_бит.\_вектор} \\
 & | & \textit{константный\_бит.\_вектор} \\
 & | & \widehat{\sim}\ \textit{терм} \\
 & | & \mathrm{ite}(\textit{терм}, \textit{терм}, \textit{терм}) \\
 & | & \textit{терм}[\textit{числовая\_константа}, .., \textit{числовая\_константа}] \\
 & | & \mathrm{sext}_{\textit{числовая\_константа}}\ \textit{терм} \\
\textit{операция} & ::= & \widehat{+} \mid \widehat{-} \mid \widehat{\times} \mid \widehat{\div}_S \mid \widehat{\mathrm{mod}}_S \mid \widehat{\ll} \mid \widehat{\gg}_U \mid \widehat{\gg}_S \mid \widehat{\&} \mid \widehat{|} \mid \widehat{\oplus} \mid \widehat{\circ}
\end{array}
$$

Здесь $\widehat{<}_S$ обозначает отношение “меньше” для знаковых целых, $\widehat{<}_U$ — для беззнаковых, $\widehat{+}$, $\widehat{-}$, $\widehat{\times}$ — сложение, вычитание и умножение по модулю 2, $\widehat{\div}_S$ и $\widehat{\mathrm{mod}}_S$ — деление и остаток от деления для знаковых целых (по модулю 2), $\widehat{\ll}$ — сдвиг влево, $\widehat{\gg}_U$, $\widehat{\gg}_S$ — логический и арифметический сдвиги вправо соответственно, $\widehat{\&}$, $\widehat{|}$, $\widehat{\oplus}$ — побитовые конъюнкция, дизъюнкция и сложение по модулю 2, $\widehat{\sim}$ — побитовое отрицание, $\widehat{\circ}$ — конкатенация битовых векторов, $\cdot[\cdot, \ldots, \cdot]$ — выделение диапазона битов, sext. $\cdot$ — знаковое расширение до заданной константной длины, $\mathrm{ite}(\cdot, \cdot, \cdot)$ — тернарный условный оператор “`если-то-иначе`”. Задача выполнимости формул в теории битовых векторов конечной длины решается с помощью редукции входной формулы к формуле в пропозициональной логике. Процесс редукции в современных решателях обычно значительно оптимизируется, например, за счет итеративного построения результирующей формулы с возрастающей точностью моделиро-

вания побитовых операций. Исходя из данных, приведенных в таблице A.1, на практике современные решатели способны разрешать формулы, содержащие в среднем несколько сотен неинтерпретируемых битовых векторов, а наибольшее их число в успешно разрешаемых формулах доходит до нескольких сот тысяч. В таблице приведены данные вне зависимости от длины битовых векторов, в тестовых наборах длины битовых векторов варьируются от 1 до 32 768, при этом средняя их длина равна 19.

A.7. Теория массивов

Теория массивов расширяет пропозициональную логику операциями над *логическими массивами*, которые являются полными отображениями из множества значений индексов во множество значений элементов. Как правило за соответствующие множества значений принимаются все возможные значения определенного типа. Таким образом, тип логического массива определяется типом его индексов и типом его элементов. Логические массивы, как правило, используются совместно с другими теориями, описывающими свойства индексов и элементов массивов. В простейшем случае в качестве теории, соответствующей индексам и элементам массивов, используется теория равенства. Теория массивов определяет для них две операции — чтения и обновления, а также отношение равенства:

$$
\begin{array}{lcl}
\textit{предикат} & ::= & \ldots \\
 & | & \textit{индекс} = \textit{индекс} \\
 & | & \textit{элемент} = \textit{элемент} \\
 & | & \textit{массив} = \textit{массив} \\
\textit{индекс} & ::= & \textit{неинтерпретируемый\_конст.\_индекс} \\
 & | & \ldots
\end{array}
$$

$$
\begin{array}{lcl}
\textit{элемент} & ::= & \textit{неинтерпретируемый\_конст.\_элемент} \\
& | & \ldots \\
\textit{массив} & ::= & \textit{неинтерпретируемая\_массивовая\_конст.} \\
& | & \textit{массив}[\textit{индекс}] \\
& | & \textit{массив}[\textit{индекс} \leftarrow \textit{элемент}]
\end{array}
$$

Операция чтения $\cdot[\cdot]$ обозначает значение элемента, сопоставляемое стоящим слева от скобок массивом заданному в скобках индексу, операция обновления $\cdot[\cdot \leftarrow \cdot]$ обозначает новый логический массив, в котором значению индекса слева от стрелки сопоставлен элемент справа от нее, а всем остальным индексам — соответствующие элементы, сопоставляемые этим индексам массивом, стоящим слева от скобок. Отношение равенства массивов *экстенсионально*, то есть массивы равны тогда и тогда, когда для каждого возможного индекса соответствующие элементы, сопоставляемые этому индексу рассматриваемыми массивами равны. Семантику определяемых теорией массивов операций и отношения равенства можно задать с использованием двух аксиом. Первая из них — основная аксиома теории массивов, определяющая семантику чтения ранее обновленного элемента (также называемая *read-over-write axiom*:

$$
\begin{gathered}
\forall a \in \mathbb{A}_{\mathbb{I}\mapsto\mathbb{E}}.\, i, j \in \mathbb{I}.\, e \in \mathbb{E}. \\
(i = j \implies a[i \leftarrow e][j] = e) \wedge \\
(i \neq j \implies a[i \leftarrow e][j] = a[j])
\end{gathered}
$$

Вторая аксиома определяет экстенсиональность массивов:

$$
\forall a, b \in \mathbb{A}_{\mathbb{I}\mapsto\mathbb{E}}.\, (\forall i \in \mathbb{I}.\, a[i] = b[i]) \implies a = b
$$

Здесь $\mathbb{I}$ — множество индексов, $\mathbb{E}$ — множество элементов, $\mathbb{A}_{\mathbb{I}\mapsto\mathbb{E}}$ — соответствующее множество массивов.

Задача о выполнимости в теории массивов является алгоритмически разрешимой [167]. Однако для формулирования большинства свойств программ,

представляющих практический интерес, безкванторной теории массивов, как правило, оказывается недостаточно.

Задача о выполнимости формул в логике первого порядка с теорией массивов в общем случае алгоритмически неразрешима [168], однако на практике часто применяют алгоритмы для разрешимых фрагментов теории массивов, например, для формул, представляющих собой пропозициональные комбинации так называемых *свойств массивов* (англ. *array properties*) — синтаксически ограниченных подформул [168]. Как видно из таблицы A.1, разрешимые на практике формулы в пропозициональной логике, расширенной только теориями равенства и массивов (логика QF_AX), содержат в среднем несколько десятков переменных-массивов. На практике теория массивов часто используется совместно с теориями вещественной и целочисленной арифметики (как линейной, так и нелинейной), а также в формулах логики первого порядка (содержащих кванторы всеобщности и существования). При использовании логики первого порядка аксиоматическое определение теории массивов позволяет перейти к рассмотрению формул в теории массивов в качестве частного случая формул первого порядка с неинтерпретируемыми функциями.

A.8. Логика первого порядка

Логика первого порядка расширяет пропозициональную логику кванторами всеобщности и существования. Соответствующее расширение грамматики пропозициональной логики:

$$\textit{предикат} \quad ::= \ \ldots \mid \exists \textit{неинт.\_константа}\,\textbf{.}\ \textit{предикат}$$

Квантор всеобщности может быть выражен с использованием квантора существования и утверждения $\forall x\,\textbf{.}\ P(x) \iff \neg\exists x\,\textbf{.}\ \neg P(x)$. Задача выполнимости логических формул первого порядка (без равенства) является PSPACE-полной [169] (и, соответственно, предположительно не входит в класс NP). В боль-

шинстве инструментов реализованы неполные разрешающие алгоритмы для логики первого порядка с равенством, задача выполнимости в которой является полуразрешимой [170, 171]. В таблице A.1 нет строки, соответствующей формулам в логике первого порядка без каких-либо теорий. Для соответствующих формул существует широко известный индекс решателей CASC [172], в котором используются задачи из набора TPTP [173] (в этом наборе также присутствуют и формулы в теории неинтерпретируемых функций), однако используемый в этом наборе формат входных формул сложно непосредственно сравнивать с формулами в форматах DIMACS и SMT-LIB по размеру входных файлов, в частности из-за широкого использования включения в формулы библиотек аксиом и нескольких различных форматов подформул. Сравнение по количеству (свободных) переменных для формул первого порядка мало показательно, так как свободные переменные легко устранимы с помощью кванторов.

A.9. Другие теории

Во многих современных SMT-решателях также реализуется поддержка разрешения логических формул в других теориях, таких как теории рекурсивных структур данных, множеств и строк. Методы моделирования семантики языка Си, рассматриваемые в рамках данной работе, не используют перечисленные теории, поэтому их краткое описание здесь не приводится, однако данные теории могут достаточно непосредственно использоваться при формализации высокоуровневых свойств Си-программ на языках спецификации, таких как ACSL. Для установления связи между заданными в этих теориях свойствами Си-программ и их семантикой, также как и для установления связи между любыми различными теориями, используемыми в рамках одной логической формулы, в решателях используются техники комбинирования

теорий.

A.10. Комбинации теорий

Комбинацией логических теорий $T_1, \ldots, T_n$ называется теория, множество функциональных и предикатных символов которой является объединением соответствующих множеств теорий $T_1, \ldots, T_n$, а семантика может быть задана с помощью объединения соответствующих наборов аксиом, задающих теории $T_1, \ldots, T_n$. Широко известны такие комбинации теорий, линейная вещественная и целочисленная арифметика с неинтерпретируемыми функциями (QF_UFLRA и QF_UFLIA соответственно), логика первого порядка с линейной вещественной арифметикой (LRA, отсутствует в таблице A.1 в силу малого числа соответствующих формул в тестовом наборе (339)), логика первого порядка с целочисленной линейной арифметикой (LIA, также известна как арифметика Пресбургера [174], представлена в SMT-COMP 201 формулой), комбинации обеих этих логик с теорией неинтерпретируемых функций (UFLRA и UFLIA), логика первого порядка с неинтерпретируемыми функциями (UF), а также логика первого порядка с теориями линейной вещественной арифметики, неинтерпретируемых функций и массивов (AUFLIRA). Известно, что среди перечисленных задача о выполнимости формул в логиках QF_UFLRA, QF_UFLIA, LRA, LIA и UF является алгоритмически разрешимой, а в логиках UFLIA и AUFLIRA — в общем случае неразрешимой.