Санкт-Петербургский государственный университет

На правах рукописи

Подкопаев Антон Викторович

# ОПЕРАЦИОННЫЕ МЕТОДЫ В ПРИЛОЖЕНИИ К СЛАБЫМ МОДЕЛЯМ ПАМЯТИ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
доктор технических наук, доцент, профессор кафедры системного программирования
КОЗНОВ Дмитрий Владимирович

Санкт-Петербург — 2018

# Содержание

Стр.

Стр.

Стр.

Стр.

# Введение

**Актуальность темы.** Многопоточные программы являются источниками специфических, трудно обнаружимых ошибок, таких как состояние гонки, взаимная блокировка и др. Эти ошибки могут воспроизводиться очень редко, например, в одном из 10 тысяч запусков программы, однако даже это может быть критично. В связи с этим необходимо иметь специализированные методы для анализа многопоточных программ. Ключевым аспектом любого анализа программ является наличие хорошо определенной семантики языка программирования. Семантики языков программирования и систем (процессоров) с многопоточностью называют *моделями памяти* (memory model, MM).

Наиболее простой и естественной моделью памяти является *последовательная консистентность* (sequential consistency, SC) [1]; она подразумевает, что каждый сценарий поведения многопоточной программы может быть получен некоторым поочередным исполнением команд её потоков на одном ядре (процессоре). Однако эта модель не способна описать все сценарии поведения, встречаемые на практике. Сценарии поведения, которые не могут быть описаны моделью SC, называются *слабыми*.

Слабыми сценариями поведения обладают некоторые многопоточные программы с неблокирующей синхронизацией потоков. Это является следствием обработки программ оптимизирующими компиляторами и их исполнением на суперскалярных процессорах. Поскольку алгоритмы на базе неблокирующей синхронизации всё чаще используются при разработке важных и высокопроизводительных систем, таких как ядро Linux и системы управления базами данных, то слабые сценарии поведения требуют тщательного изучения.

Модели памяти, допускающие слабые сценарии поведения программ, называются *слабыми* (weak memory models) [2]. На данный момент научное сообщество в тесном сотрудничестве с индустрией разработало и продолжает совершенствовать множество таких моделей для языков программирования и процессорных архитектур. При этом процессорные и языковые модели существенно влияют на друг друга. Так, модель процессора должна отражать сценарии поведения, наблюдаемые при запуске программ на существующих процессорах, и оставлять возможности для развития обратно совместимых архитектур. В то же время, языковая модель должна предоставлять разумные и удобные абстракции для программиста, а также допускать основные компиляторные оптимизации и быть совме-

стимой с моделям целевых архитектур, т.е. поддерживать эффективную трансляцию в низкоуровневый код без изменения семантики программы.

Существующие модели памяти для наиболее популярных языков программирования обладают рядом недостатков. Так, известно, что модель памяти Java [3] некорректна по отношению к базовым оптимизациям, а модель памяти C/C++11 [4] разрешает сценарии поведения программ, в которых появляются “значения из воздуха” (out-of-thin-air values) [5]. Модель памяти имеет проблему “значений из воздуха”, если для программы без арифметики, в которой явным образом не встречается некоторая константа, (например, 42) допустим сценарий поведения, в котором эта константа появляется (например, записывается в память или читается из памяти). Такие сценарии не проявляются на практике, но тот факт, что модель C/C++11 их допускает, не позволяет формально доказывать многие полезные свойства программ в рамках этой модели. Наличие проблемы “значений из воздуха” связано с тем, что модель C/C++11 задана декларативно (аксиоматически), при этом сценарий поведения программы в рамках модели определяется как некоторая монолитная структура (граф), а не как последовательность действий некоторой абстрактной машины. Это оставляет открытым вопрос об интеграции модели с остальными компонентами языка, которые в стандарте C/C++11 определены операционно.

Таким образом, для развития инструментов анализа многопоточных программ необходимо разработать операционные подходы к заданию слабых моделей памяти.

**Степень разработанности темы.** С 1990-х годов велась работа по разработке семантики многопоточности с учетом слабых сценариев поведения. Формальные модели для наиболее распространенных процессорных архитектур (x86, Power, ARM) были разработаны J. Alglave, S. Ishtiaq, L. Maranget, F. Zappa Nardelli, S. Sarkar, P. Sewell и др. исследователями [6–8]. Новые версии моделей продолжают появляться в связи с развитием процессорных архитектур. В частности, в 2016 и 2017 годах были представлены модели памяти для архитектур ARMv8.0 и ARMv8.3 [9, 10]. В 1995 году была стандартизована слабая модель памяти для языка Java [3]; в дальнейшем модель существенно менялась вплоть до 2005 года. В 2011 году появилась аксиоматическая модель памяти для языков C/C++ [4].

В 2017 году исследователи J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis и D. Dreyer представили обещающую модель памяти (promising memory model,

Promise) [11], которая является перспективным решением проблемы задания семантики для языка с многопоточностью. Авторы доказали, что модель допускает большинство необходимых оптимизаций, а также показали корректность эффективных схем компиляции в архитектуры x86 и Power. Открытым остался вопрос о корректности компиляции в архитектуру ARM.

**Целью** данной работы является исследование применимости операционных подходов для описания реалистичных моделей памяти и анализа многопоточных программ на примере языков C/C++.

Для достижения поставленной цели были сформулированы следующие **задачи**.

1. Разработать операционную модель памяти C/C++11, свободную от проблемы “значений из воздуха”.
2. Доказать корректность эффективной схемы компиляции из существенного подмножества обещающей модели в операционную модель памяти ARMv8 POP.
3. Доказать корректность эффективной схемы компиляции из существенного подмножества обещающей модели в аксиоматическую модель памяти ARMv8.3.

Постановка цели и задач исследования соответствует следующим пунктам паспорта специальности 05.13.11: модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования (пункт 1); языки программирования и системы программирования, семантика программ (пункт 2); модели и методы создания программ и программных систем для параллельной и распределенной обработки данных, языки и инструментальные средства параллельного программирования (пункт 8).

**Методология и методы исследования.** Методология исследования базируется на подходах информатики к описанию и анализу формальных семантик языков программирования.

В работе используется представление операционной семантики программы с помощью помеченной системы переходов [12], а также метода вычислительных контекстов, предложенного M. Felleisen [13]. Для доказательств корректности компиляции используется техника прямой симуляции [14, 15]. Программная реализация интерпретатора операционной модели памяти C/C++11 выполнена на

языке Racket [16,17] с использованием предметно-ориентированного расширения PLT/Redex [18, 19].

**Основные положения, выносимые на защиту.**

1. Предложена операционная модель памяти C/C++11, для этой модели реализован интерпретатор.
2. Доказана корректность компиляции из существенного подмножества обещающей модели в операционную модель памяти ARMv8 POP.
3. Доказана корректность компиляции из существенного подмножества обещающей модели в аксиоматическую модель памяти ARMv8.3.

**Научная новизна** результатов, полученных в рамках исследования, заключается в следующем.

1. Альтернативная модель памяти для стандарта C/C++11, предложенная в работе, отличается от обещающей модели памяти [11] тем, что является запускаемой, т.е. для нее возможно создание интерпретатора (что и было выполнено в рамках данной диссертационной работы). Это отличие является следствием того, что для получения эффекта отложенного чтения предложенная модель использует синтаксический подход (буферизация инструкций), тогда как обещающая модель — семантический (обещание потоком сделать запись в будущем).
2. Доказательство корректности компиляции из обещающей модели памяти в аксиоматическую модель ARMv8.3 [10] не опирается на специфические свойства целевой модели, такие как представимость модели в виде набора оптимизаций поверх более строгой модели. Это отличает его от аналогичных доказательств для моделей x86 и Power (работы O. Lahav, V.Vafeiadis и других [11, 20]).
3. Доказательства корректности компиляции из обещающей модели памяти в модели ARMv8 POP [9] и ARMv8.3 [10], представленные в работе, являются первыми результатами о компиляции для данных моделей.

**Теоретическая и практическая значимость работы.** Диссертационное исследование предлагает новый операционный способ представления реалистичной семантики многопоточности с помощью меток времени и фронтов, который может быть полезен при верификации многопоточных программ с неблокирующей синхронизацией, а также при анализе реализации примитивов блокирующей синхронизации.

Предложенный в диссертационном исследовании метод доказательства корректности компиляции из обещающей в аксиоматические модели памяти может быть использован для доказательств корректности компиляции из обещающей модели в архитектуры других процессоров. Последнее актуально в свете того, что в комитетах по стандартизации языков C и C++ активно обсуждается вопрос о смене модели памяти, и обещающая модель является одной из возможных альтернатив.

**Степень достоверности и апробация результатов.** Достоверность и обоснованность результатов исследования обеспечивается формальными доказательствами, а также инженерными экспериментами. Полученные результаты согласуются с результатами, установленными другими авторами.

Основные результаты работы докладывались на следующих научных конференциях и семинарах: внутренний семинар ННГУ им. Лобачевского (13 декабря 2017, Нижний Новгород, Россия), открытая конференция ИСП РАН им. В.П. Иванникова (30 ноября–1 декабря 2017, РАН, Москва, Россия), семинар “Технологии разработки и анализа программ” (16 ноября 2017, ИСП РАН, Москва, Россия), внутренние семинары School of Computing of the University of Kent (август 2017, Кентербери, Великобритания), внутренние семинары Department of Computer Science of UCL (август 2017, Лондон, Великобритания), внутренние семинары MPI-SWS (май 2017, Кайзерслаутерн, Германия), The European Conference on Object-Oriented Programming (ECOOP, 18–23 июня 2017, Барселона, Испания), конференция “Языки программирования и компиляторы” (PLC, 3–5 апреля 2017, Ростов-на-Дону, Россия), Verified Trustworthy Software Systems workshop (VTSS, 4-7 апреля 2016, Лондон, Великобритания), POPL 2016 Student Research Competition (21 января 2016, Санкт-Петербург, Флорида, США).

**Публикации по теме диссертации.** Основные результаты по теме диссертации изложены в пяти печатных работах, зарегистрированных в РИНЦ. Из них две статьи изданы в журналах из “Перечня рецензируемых научных изданий, в которых должны быть опубликованы основные научные результаты диссертаций на соискание ученой степени кандидата наук, на соискание ученой степени доктора наук”, сформированного согласно требованиям, установленным Министерством образования и науки Российской Федерации. Одна статья опубликована в издании, входящем в базы цитирования SCOPUS и Web of Science.

Личный вклад автора в публикациях, выполненных с соавторами, распределён следующим образом. В статьях [21,22] автор предложил схему доказательства

корректности компиляции для аксиоматических семантик и выполнил само доказательство для модели ARMv8.3; соавторы участвовали в обсуждении основных идей доказательства. В работах [23,24] автор выполнил формализацию семантики ARMv8 POP и доказал корректность компиляции методом симуляции; соавторы участвовали в обсуждении идей доказательства и редактировали тексты статей. В работе [25] личный вклад автора заключается в предложении идеи меток времени и фронтов как способа операционного задания модели памяти, а также в создании компонентного метода задания семантики и реализации интерпретатора; соавторы предложили синтаксический способ обработки отложенных операций.

**Благодарности.**

Я хочу выразить признательность своим коллегам, друзьям и семье, присутствие которых в моей жизни сделало эту работу возможной. Начать я хотел бы с выражения благодарности Д. Ю. Булычеву, А. В. Иванову и компании JetBrains, которые предоставили мне уникальную возможность заниматься наукой как основной деятельностью. Я признателен своему научному руководителю Д. В. Кознову, который помог мне пройти через тернистый путь создания диссертации. Я благодарен И. Д. Сергею и А. Наневски (A. Nanevski) за бескорыстную поддержку и руководство моей работой на первых этапах. Я хотел бы выразить признательность старшим коллегам В. Вафеядису (V. Vafeiadis) и О. Лахаву (O. Lahav) за помощь в исследованиях и сотрудничество. Я очень благодарен коллективам MPI-SWS, IMDEA Software и кафедры системного программирования, а также лично А. Н. Терехову за создание душевной и плодотворной атмосферы для работы.

Я признателен за удивительную отзывчивость и дружбу Д. А. Березуну, за помощь в тяжелейших ситуациях и человеческую доброту А. М. Карачуну, за немыслимые поддержку и понимание А. В., Т. П. и В. Е. Турецким и моей семье. Я горячо признателен моей маме за её самоотверженное стремление помочь мне на каждом этапе моей жизни. Я безмерно благодарен моему папе за всё.

# Глава 1. Обзор предметной области

В данной главе вводятся основные понятия, использующиеся в этом диссертационном исследовании: модель памяти, слабая модель памяти и др. — а также приводятся примеры. Рассматриваются существующие модели памяти языков программирования и процессорных архитектур, а также требования, предъявляемые к ним. Приводится описание проблемы “значений из воздуха” (OOTA, out-of-thin-air values). Подробно описывается модель памяти C/C++11 [4]. В конце главы приведены выводы о состоянии предметной области и о существующих открытых проблемах.

## 1.1 Модели памяти и слабые сценарии поведения

В диссертационном исследовании под *моделью памяти* понимается семантика системы с многопоточностью, и рассматриваются два типа таких систем: языки программирования и процессорные архитектуры.

Модели памяти разделяются по принципу того, какие ограничения на сценарии поведения программ они накладывают [26]. Так, *строгая консистентность* гарантирует, что любая запись в память становится мгновенно видна всем потокам в системе. Эта модель требует наличия некоторого абсолютного счётчика времени, разделяемого всеми потоками системы, что зачастую является недостижимым. Менее строгая модель *последовательной консистентности* [1] (SC, sequential consistency) предполагает, что любой сценарий поведения может быть получен исполнением потоков на одном вычислительном ядре с вытесняющей многозадачностью. Это означает, что все операции над памятью, совершаемые потоками в рамках одного сценария поведения, могут быть упорядочены, и полученный порядок согласуется с порядком инструкций в самих потоках. Сценарии поведения программ, которые не могут быть получены в рамках модели SC, называются *слабыми*, а модели, допускающие слабые сценарии поведения, — *слабыми моделями памяти* [2].

Несмотря на то, что модель SC кажется наиболее естественной, а статья [1], в которой эта модель описывается, называется “How to make a multiprocessor computer that correctly executes multiprocess programs”, современные процессорные архитектуры и языки программирования активно используют слабые модели памяти. Это связано с тем, что такие модели позволяют реализовать большее

число оптимизаций как на уровне процессора [27], так и на уровне компилятора [28, 29], что увеличивает производительность программ.

Рассмотрим следующую программу MP (message passing, передача сообщения):

$$\begin{array}{lll} [x] := 0; & [y] := 0; & \\ [x] := 1; & a := [y]; & //1 \\ [y] := 1 & b := [x] & //0 \end{array} \qquad \text{(MP)}$$

Эта программа является упрощенным примером передачи данных между потоками. Первый поток записывает данные в локацию $x$ и потом выставляет флаг (локация $y$), что данные подготовлены; в свою очередь второй поток проверяет этот флаг, а потом читает данные. Модель SC гарантирует, что если второй поток увидел, что флаг выставлен ($a = 1$), то он увидит и подготовленные данные ($b = 1$). Тем не менее, эта программа имеет слабый сценарий поведения с результатом $[a = 1, b = 0]$ на таких архитектурах как Power и ARM.

С чем связано то, что на упомянутых выше архитектурах возможен результат $[a = 1, b = 0]$? Оптимизирующий процессор при исполнении программы может выполнить независимые инструкции не по порядку. Поскольку первая и вторая инструкции в левом потоке являются обращениями к разным локациям, то процессор может выполнить сначала вторую запись, а потом первую. То же самое верно и для инструкций в правом потоке. После исполнения инструкций не по порядку хотя бы в одном потоке результат $[a = 1, b = 0]$ становится возможным.

Такой сценарий поведения также разрешается моделями памяти некоторых языков программирования, например, стандартами языков C11 [30] и C++11 [31], поскольку оптимизирующий компилятор должен иметь возможность переупорядочить независимые обращения к памяти.

## О корректности программы MP. Гонки по данным

С точки зрения некоторых языков программирования программа MP может считаться некорректной, т.к. в этой программе есть *гонка по данным* [32].

**Определение 1.** В программе имеется *гонка по данным* (data race), если в некотором её сценарии поведения существуют два неупорядоченных обращения к одной и той же ячейке памяти, причём, как минимум, одно из этих обращений является операцией записи.

Данное определение не является формальным, т.к. здесь не определяется порядок на операциях над памятью. Это связано с тем, что в разных моделях этот порядок определяется существенно по-разному. По сути, два обращения неупорядочены, если порядок их исполнения определяется не логикой программы, а внешними факторами, такими как, например, диспетчеризация потоков.

В отсутствии гонок по данным большинство слабых моделей памяти гарантируют, что все сценарии поведения являются SC-поведениями. Для того, чтобы добиться отсутствия гонок даже при использовании общей памяти (shared memory), применяют *блокировки* (locks), которые упорядочивают обращения к разделяемому ресурсу.

Как следствие, один из способов задать модель памяти для языка программирования выглядит следующим образом: если в программе нет гонок по данным, то её поведение определяется моделью SC, иначе программа является некорректной и обладает неопределенным поведением (undefined behavior). У такого способа есть, как минимум, два недостатка. Во-первых, языки программирования (например, Java), стремясь обеспечить *типобезопасность* (type safety), не могут использовать такой способ задания семантики, т.к. гарантируют, что программа не может иметь неопределенное поведение, если она прошла проверку типов, а наличие или отсутствие гонок по данным не может быть проверено статически. Во-вторых, многие высокопроизводительные алгоритмы многопоточного программирования используют парадигму *неблокирующей синхронизации* (non-blocking synchronization), которая существенным образом опирается на гонки по данным. Более того, в большинстве случаев реализация самих блокировок использует гонки по данным, что делает невозможным рассуждения о ней в рамках приведенной выше упрощённой модели.

Из вышесказанного следует, что модель памяти промышленного языка программирования должна обеспечивать корректную семантику, как минимум, для некоторого множества программ с гонками по данным.

## 1.2 Требования к моделям памяти

На данный момент имеется множество моделей памяти как для процессорных архитектур [6, 7, 9, 10, 33, 34], так и для языков программирования [1, 3, 4, 35–40]. Существенным отличием между этими группами моделей являются предъявляемые к ним требования.

Модели процессорных архитектур должны описывать сценарии поведения существующих процессоров, а также оставлять пространство для возможных оптимизаций для следующих версий процессоров. Кроме того, такие модели зачастую либо заданы *операционно*, т.е. в терминах некоторой абстрактной машины [41], либо имеют эквивалентное операционное представление. Это позволяет определить модель в терминах, близких и понятных разработчикам архитектуры, а также дать разработчикам компиляторов интуитивно понятное представление об исполнении программы.

Модель памяти языка программирования должна быть представлена таким образом, чтобы, с одной стороны, она разрешала манипуляции над кодом программы, совершаемые в рамках компиляторных оптимизаций, и давала возможность эффективно компилировать программы в целевую процессорную архитектуру, а, с другой стороны, предоставляла разумные гарантии для программиста. Эти требования до некоторой степени противоречат друг другу, поэтому хорошая модель соблюдает баланс между ними. Для того, чтобы понять, в чём заключается противоречие данных требований, остановимся на них подробнее.

### 1.2.1 Корректность компиляторных оптимизаций

Пусть есть некоторый язык программирования $\mathsf{L}$[1] и его модель памяти $\mathcal{M}$ (memory model). Тогда под *семантикой программы* $\mathsf{P}$ *на языке* $\mathsf{L}$ *в модели* $\mathcal{M}$ будет пониматься множество возможных сценариев поведения $\mathsf{P}$ в $\mathcal{M}$. Это множество будет обозначаться $[\![\mathsf{P}]\!]_{\mathcal{M}}$. *Оптимизацией над программами в языке* $\mathsf{L}$ мы будем называть функцию $\mathsf{opt}$, действующую из множества программ на языке $\mathsf{L}$ в него же.

**Определение 2.** Оптимизация $\mathsf{opt} : \mathsf{L} \to \mathsf{L}$ называется *корректной в модели* $\mathcal{M}$, $\mathsf{CorrectOpt}_{\mathcal{M}}(\mathsf{opt})$, если для любой программы $\mathsf{P}$ на языке $\mathsf{L}$ семантика оптимизированной программы $\mathsf{opt}(\mathsf{P})$ является подмножеством семантики изначальной программы $\mathsf{P}$:

$$\forall \mathsf{opt} : \mathsf{L} \to \mathsf{L}.\ \mathsf{CorrectOpt}_{\mathcal{M}}(\mathsf{opt}) \Leftrightarrow (\forall \mathsf{P} \in \mathsf{L}.\ [\![\mathsf{opt}(\mathsf{P})]\!]_{\mathcal{M}} \subseteq [\![\mathsf{P}]\!]_{\mathcal{M}}).$$

Какие компиляторные оптимизации должны быть корректны в рамках модели памяти языка программирования? К сожалению, на данный момент не су-

[1] Здесь язык программирования рассматривается как множество высказываний (в нашем случае — программ) на этом языке.

ществует полного списка таких оптимизаций, однако из работ [42–44] можно выделить пять основных групп.

1. Локальные оптимизации, не меняющие обращения к памяти. Например, удаление условных переходов, зависимых от заведомо ложного условия:

$$\begin{array}{l} a := 0; \\ \textbf{if}\ a \\ \textbf{then}\ b := [x] \\ \textbf{else skip} \\ \textbf{fi}; \\ c := [y] \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a := 0; \\ \textbf{skip}; \\ c := [y] \end{array}$$

2. Перестановка независимых обращений к памяти. Например, перестановка инструкций чтения из разных локаций:

$$\begin{array}{l} a := [x]; \\ b := [y] \end{array} \quad \rightsquigarrow \quad \begin{array}{l} b := [y]; \\ a := [x] \end{array}$$

3. Устранение избыточных обращений к памяти. Например, устранение инструкции чтения, следующей за инструкциями чтения или записи в ту же локацию:

$$\begin{array}{l} a := [x]; \\ b := [x] \end{array} \rightsquigarrow \begin{array}{l} a := [x]; \\ b := a \end{array} \qquad \begin{array}{l} [x] := a; \\ b := [x] \end{array} \rightsquigarrow \begin{array}{l} [x] := a; \\ b := a \end{array}$$

4. Вставка избыточных обращений к памяти. Например, вставка инструкции чтения в переменную, значение которой далее нигде не используется.
5. Глобальные оптимизации. Например, *секвенциализация* (sequentialization), которая заменяет параллельную композицию потоков на последовательную:

$$C_1 \,||\, C_2 \;\rightsquigarrow\; C_1;\, C_2$$

Желательно, чтобы упомянутые классы оптимизаций были корректными, возможно, с некоторыми оговорками в модели памяти языка программирования. Для этого модель должна быть достаточно слабой, т.е. позволять сценарии поведения, которые возможны для программы после оптимизации.

### 1.2.2 Наличие эффективной схемы компиляции

Модель памяти языка программирования должна учитывать модель памяти целевой платформы, т.е. должна существовать корректная схема компиляции из одной модели в другую.

**Определение 3.** Пусть есть некоторые языки $\mathsf{L}$ и $\mathsf{L}'$ и соответствующие модели памяти $\mathcal{M}$ и $\mathcal{M}'$. Функция $\mathsf{compl} : \mathsf{L} \to \mathsf{L}'$ является *корректной схемой компиляции из модели $\mathcal{M}$ в модель $\mathcal{M}'$*, если для любой программы $\mathsf{P}$ на языке $\mathsf{L}$ семантика программы $\mathsf{compl}(\mathsf{P})$ в модели $\mathcal{M}'$ является подмножеством семантики $\mathsf{P}$ в модели $\mathcal{M}$.

Из этого определения следует, что чем слабее модель целевой платформы, т.е. чем больше существует сценариев поведения на целевой платформе, тем больше ограничений накладывается на корректную схему компиляции.

Рассмотрим то, как должна быть устроена корректная схема компиляции из более строгой модели памяти в более слабую на примере компиляции ранее приведенной программы MP из модели SC [1] в модель архитектуры Power [6]. В рамках модели SC программа MP не имеет сценария поведения $[a = 1, b = 0]$, тогда как в модели Power такой сценарий возможен. Для того, чтобы получить корректную компиляцию, в скомпилированную программу нужно вставить специальные инструкции — т.н. *барьеры памяти*. Эти барьеры вносят дополнительные ограничения на сценарии поведения программ. Достигается это за счёт того, что барьеры запрещают некоторые компиляторные и процессорные оптимизации. В архитектуре Power есть барьер hwsync, который запрещает переупорядочивание любых инструкций вокруг него. Вставка такого барьера между инструкций в программе MP гарантирует отсутствие сценария поведения $[a = 1, b = 0]$ в модели Power[2]:

$$\begin{array}{l|l} [x] := 0; & [y] := 0; \\ [x] := 1; & a := [y]; \\ \mathsf{hwsync}; & \mathsf{hwsync}; \\ [y] := 1 & b := [x] \end{array} \qquad \text{(MP-hwsync)}$$

На ряду с hwsync архитектура Power также предоставляет более слабый барьер lwsync, который запрещает только перестановки пар инструкций чтение-чтение,

[2]Здесь и далее в диссертации используется один и тот же синтаксис для описания как исходных, так и скомпилированных программ с точностью до барьеров и модификаторов чтения и записи.

чтение-запись и запись-запись. Такого барьера также достаточно, чтобы запретить сценарий $[a = 1, b = 0]$:

$$\begin{array}{l||l} [x] := 0; & [y] := 0; \\ [x] := 1; & a := [y]; \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ [y] := 1 & b := [x] \end{array} \qquad \text{(MP-lwsync)}$$

В данном случае схема компиляции, использующая барьер lwsync, является более предпочтительной по сравнению со hwsync-схемой, т.к. барьер lwsync на реальных процессорах исполняется быстрее (или, как минимум, не медленнее), чем hwsync.

Если рассмотреть модель памяти языка программирования, которая разрешает сценарий поведения с результатом $[a = 1, b = 0]$ (на ряду со всеми остальными сценариями, возможными в рамках модели SC), то для компиляции программы MP из такой модели в модель Power не будет необходимости вставлять барьеры памяти. Это хорошо, т.к. скомпилированная программа может быть эффективно исполнена на целевом процессоре. Так, модели C/C++11 [4] и Java [3] разработаны таким образом, чтобы обычные операции чтения и записи[3] могли быть скомпилированы без вставки барьеров памяти в случае компиляции в платформы x86, Power и ARM.

Подобные схемы компиляции будут называться *эффективными* в рамках данного исследования. Соответственно, как было видно из рассмотренных схем компиляции из модели SC в модель Power, для этих моделей не существует эффективной компиляции. Очевидно, что для того, чтобы существовала эффективная схема компиляции из модели языка программирования в модель целевой платформы, должно выполняться следующее утверждение: модель исходного языка должна позволять все сценарии поведения для программы без барьеров, которые позволяет целевая модель памяти для скомпилированной версии программы, в которой также не используются барьеры. Аналогично требованию на корректность оптимизаций, наличие эффективной схемы компиляции увеличивает число слабых поведений, допускаемых моделью языка программирования.

[3] В разделе 1.3.3 будут рассмотрены различные модификаторы, которыми могут быть помечены инструкции работы с памятью в модели C/C++11, и “обычные операции чтения и записи” будут соответствовать атомарным расслабленным и неатомарным инструкциям. Все инструкции работы с памятью из примеров программ, приведённых к этому моменту, были “обычными”.

### 1.2.3 Гарантии программисту

Требование о том, что модель памяти языка программирования должна предоставлять разумные гарантии программисту звучит крайне неформально. Попробуем сформулировать его более конкретно.

Очевидно, что программист должен иметь представление о том, как ведется себя программа, над которой он работает, а поведение программы как раз определяется моделью памяти. Как следствие, желательно, чтобы модель памяти была максимально простой и понятной, а также существовал формальный инструментарий для рассуждения о программах в рамках модели. Простота — это очень субъективный критерий, особенно при сравнении слабых моделей памяти, тогда как наличие и выразительная сила инструментов анализа может быть использована для более точного сопоставления моделей.

Существуют различные методы для анализа программ. Среди них стоит отметить методы *проверки моделей* (model checking) [45, 46] и *верификации по Хоару* (Hoare logic) [47]. Оба метода хорошо зарекомендовали себя в приложении к модели SC [48–63]. Существуют также работы, посвященные слабым моделям памяти [64–69]. В последних отмечается, что если модель памяти разрешает сценарии поведения программ с т.н. *“значениями из воздуха”* (out-of-thin-air values, OOTA), то класс свойств, которые могут быть проверены или доказаны для программ в рамках таких моделей, существенным образом ограничивается. В частности, модель памяти C/C++11 разрешает такие сценарии.

Что же такое “значения из воздуха”? К сожалению, на данный момент не существует точного определения [5], однако существует набор примеров таких поведений, а также следующий признак наличия “значений из воздуха”. Если в сценарии поведения программы, в которой нет арифметических выражений, появляется некоторое значение (например, это значение присваивается переменной или записывается в память), причём это значение явным образом не фигурирует в тексте программы, то такой сценарий обладает “значениями из воздуха”.

Рассмотрим следующую программу LB-OOTA (load buffering, буферизация чтения):

$$\begin{array}{l|l} [x] := 0; & [y] := 0; \\ a := [x]; & b := [y]; \\ [y] := a & [x] := b \end{array} \qquad \text{(LB-OOTA)}$$

В этой программе нет арифметических операций и $0$ является единственным значением, которое явным образом встречается в тексте программы. По приведенному выше признаку любой сценарий поведения этой программы, в результате которого значение переменной $a$ или $b$ отличается от $0$, обладает “значением из воздуха”. Такие сценарии разрешаются для этой программы в рамках модели C/C++11 [4].

Рассмотрим другую программу, некоторые сценарии поведения которой в рамках модели C/C++11 также считаются [5] обладающими “значениями из воздуха”, но не подпадающую под сформулированный выше признак:

$$
\begin{array}{l}
[x] := 0; [y] := 0; \\
\begin{array}{l|l}
a := [x]; & b := [y]; \\
\textbf{if } a & \textbf{if } b \\
\textbf{then } [y] := 1 & \textbf{then } [x] := 1 \\
\textbf{else skip} & \textbf{else skip} \\
\textbf{fi} & \textbf{fi}
\end{array}
\end{array}
\qquad \text{(IF-OOTA)}
$$

Казалось бы, что единственным результатом исполнения этой программы может быть $[a = 0, b = 0]$, но модель C/C++11 также разрешает $[a = 1, b = 1]$.

Сценарии поведения со “значениями из воздуха” всегда имеют некоторую циклическую зависимость между встречаюшимися в них значениями. Как следствие, они не проявляются на современных процессорах[4] и не могут быть получены как результат разумной оптимизации. То, что некоторые модели разрешают такие сценарии, обычно является результатом того, что в рамках подхода, выбранного для задания модели, тяжело или невозможно запретить “значения из воздуха” и, при этом, не запретить нужные оптимизации [71].

## 1.3 Существующие модели памяти языков программирования

Среди моделей памяти, разработанных для языков программирования, есть как используемые на практике, т.е. являющиеся частями стандартов языков [30, 31, 72, 73], так и теоретические [1, 35–39]. Далее в этом разделе обсуждаются модели памяти языков Java [73] и C/C++ [30,31], выделяются их достоинства и недостатки, а также рассматривается, как упомянутые недостатки решаются в существующих теоретических моделях.

[4]Сценарии поведения со “значениями из воздуха” проявляются на процессорах семейства DEC Alpha [70], но данная архитектура снята с производства.

### 1.3.1 Виды моделей памяти

На данный момент существует два основных подхода к описанию моделей памяти — *аксиоматический* (декларативный) и *операционный*, которые в некоторых работах смешиваются. Аксиоматические модели памяти представляют сценарии поведения программ в виде графов, которые должны соответствовать определенным аксиомам. Модели памяти Java и C/C++, описанные ниже, являются примерами аксиоматических моделей. Преимуществами аксиоматических моделей является относительная простота для задания глобальных свойств сценария поведения программы, таких как наличие некоторого тотального порядка над подмножестве событий (вершин графа) определенного вида. Также аксиоматические модели проще для представления в логических языках, таких как Alloy [74] и Rosette [75, 76], что используется для автоматического сравнения моделей [77] и генерации моделей по набору примеров [78].

Сценарий поведения в рамках операционных моделей представляется как серия переходов некоторой абстрактной машины. Преимуществом таких моделей является то, что они дают представление об исполнении программы. Как следствие, между моделью памяти, которая обычно определяет поведение только подмножества языка, связанное с его многопоточной компонентой, и описанием остальных конструкций языка имеется прослеживаемая связь, которую нужно дополнительно устанавливать для аксиоматических моделей [79]. Также для большинства операционных моделей возможно разработать интерпретатор, что позволяет пошагово отлаживать многопоточные алгоритмы.

### 1.3.2 Модель памяти Java

Впервые модель памяти для языка Java была представлена в стандарте 1996 года [72]. Эта модель обладала рядом фундаментальных недостатков [80,81] и была заменена на новую модель [3], которая дальше будет упоминаться как модель памяти Java, или JMM (Java memory model). Основной задачей при разработке JMM было разработать модель без “значения из воздуха”, но разрешить базовые компиляторные оптимизации. К сожалению, JMM не удовлетворяет последнему требованию [44]: в рамках JMM некорректными являются оптимизации удаление чтения после чтения (redundant read after read elimination), удаление чтения после записи (redundant read after write elimination), удаление записи после чте-

ния (redundant write after read elimination), добавление неиспользуемого чтения (irrelevant read introduction) и другие.

Сценарий поведения программы в JMM представляется как граф, в котором вершинами являются *события* (actions, events), происходящие в памяти, а помеченными ребрами — отношения на событиях. События бывают нескольких типов, из которых основными являются: чтение, запись, захват замка (lock), высвобождение замка (unlock). События чтения и записи, в свою очередь, бывают обычные (non-volatile) и *синхронизирующие* (volatile). Первые действуют на обычных локациях в памяти (переменных), тогда как синхронизирующие – на *изменчивых* (volatile) локациях. Изменчивые локации выделены как локации, посредством которых происходит синхронизация.

Отметим пять отношений, используемых в сценариях поведения JMM. Первое отношение, *программный порядок* (program order, $\texttt{po}$), для каждого потока является тотальным порядком на событиях, относящихся к этому потоку. Второе отношение, *"читает из"* (reads from, $\texttt{rf}$), связывает событие записи с читающими из него событиями. Третье отношение, *синхронизационный порядок* (synchronization order, $\texttt{so}$), является тотальным порядком на синхронизирующих событиях и событиях захвата и высвобождения замков; оно должно быть согласовано с отношениями $\texttt{po}$ и $\texttt{rf}$ на синхронизирующих событиях. Четвёртое отношение, *"синхронизируется с"* (synchronizes-with, $\texttt{sw}$), связывает события $a$ и $b$, если $\texttt{rf}$ связывает $a$ и $b$ и эти события действуют над изменчивой локацией, или если они связаны с помощью $\texttt{so}$ и $a$ является высвобождением некоторого замка, а $b$ — захватом этого же замка. Пятое отношение, *"предшествует"* (happens-before, $\texttt{hb}$), является транзитивным замыканием объединения отношений $\texttt{po}$ и $\texttt{sw}$; именно это отношение определяет порядок на событиях из определения 1 для JMM.

Сценарий поведения в JMM считается корректным, если для него выполняются *аксиомы* модели. Одной из аксиом является то, что событие чтения не может читать из события записи, если они связаны отношением $\texttt{hb}$ и существует событие записи в ту же локацию, которое находится между ними в отношении $\texttt{hb}$. Формально это можно выразить следующим образом:

$$\begin{aligned}&\forall r, w.\ \texttt{rf}(w, r) \wedge \texttt{hb}(w, r) \Rightarrow \\ &\quad \nexists w'.\ \texttt{loc}(w) = \texttt{loc}(w') \wedge \texttt{hb}(w, w') \wedge \texttt{hb}(w', r),\end{aligned}$$

где $\texttt{loc}$ — это функция, которая по событию возвращает локацию, над которой событие оперирует. В силу этой аксиомы JMM запрещает результат $[a = 1, b = 0]$

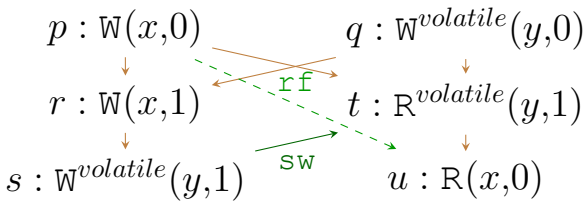



Рис. 1. Сценарий программы MP-volatile, который не соответствует аксиомам модели JMM

для версии программы MP, в которой локация $y$ помечена как изменчивая:

$$\begin{array}{lll} [x] := 0; & [y] := 0; & \\ [x] := 1; & a :=_{\text{volatile}} [y]; & //1 \\ [y] :=_{\text{volatile}} 1 & b := [x] & //0 \end{array} \qquad \text{(MP-volatile)}$$

Для того, чтобы проверить предыдущее утверждение, рассмотрим соответствующий сценарий поведения (см. рис. 1). В его графе имеется шесть событий. События $p$ и $q$ являются инициализирующим записям в локации $x$ и $y$, а остальные — инструкциям из MP-volatile. Недописанные дуги — это дуги отношения $\mathtt{po}$. На рис. 1 опущены дуги отношения $\mathtt{hb}$, связывающие $p$ и $r$ ($\mathtt{po} \subseteq \mathtt{hb}$), а также $r$ и $u$ (т.к. $\mathtt{po}(r,s)$, $\mathtt{sw}(s,t)$, $\mathtt{po}(t,u)$ и $\mathtt{hb} = (\mathtt{po} \cup \mathtt{sw})^+$). Таким образом, данный граф противоречит приведенной выше аксиоме и не является корректным в JMM.

Отметим, что если бы локация $y$ не была бы помечена как изменчивая, то события $s$ и $t$ имели бы обычный тип и между ними бы не было ребра $\mathtt{sw}$, следовательно аксиома бы выполнялась и результат $[a = 1, b = 0]$ был бы возможен.

Для того, чтобы запретить “значения из воздуха”, в JMM используется процесс валидации сценариев. В рамках этого процесса строится серия сценариев, каждый из которых является корректным с точки зрения аксиом JMM, при этом последним в серии является валидируемый сценарий. Также каждому сценарию в серии присваивается множество добавленных (committed) событий. Эти множества монотонно растут, а последнее является множеством всех событий в графе валидируемого сценария. При этом на каждом шаге трансформации в серии сохраняется подмножество отношения $\mathtt{hb}$, заданное на добавленных событиях.

В [44] показывается, что именно из-за процесса валидации упомянутые выше оптимизации некорректны в JMM.

### 1.3.3 Модель памяти C/C++

В последние годы научное сообщество уделяет особое внимание модели памяти C/C++, которая появилась в стандартах языков в 2011 году [30, 31]. Она не претерпела существенных изменений в последующих стандартах 2014 [82] и 2017 [83] годов, и поэтому в литературе обычно не различают понятия “модель памяти C/C++” и “модель памяти C/C++11”. Далее в работе используется “модель памяти C/C++11” или “C/C++11 MM”.

Базовым результатом о свойствах этой модели является её формализация, представленная в [4]; далее под C/C++11 MM будет подразумеваться именно эта формализация. Существуют работы, посвященные корректности компиляции и оптимизаций [42, 84–87], логикам для формальных рассуждений о программах [65–67, 88, 89] и другим свойствам этой модели [71, 90–93]. Как следствие, C/C++11 MM является одной из самых проработанных моделей памяти для языков программирования. Тем не менее, она не лишена недостатков, в том числе “значений из воздуха”.

#### Виды обращений к памяти

В C/C++11 MM локации бывают двух типов — неатомарные (non-atomic) и атомарные (atomic). В отличие от языка Java, разделение между ними менее строгое, т.к. языки C/C++ разрешают преобразование типов, следовательно на один и тот же адрес в памяти может ссылаться две переменных в программе, одна из которых будет иметь атомарный тип, а другая — неатомарный. Поэтому в C/C++11 MM обращения к одной и той же локации могут быть как неатомарными, так и атомарными. Последние, в свою очередь, также подразделяются на несколько категорий в зависимости от типа обращения — чтение, запись или одновременное чтение-запись (RMW, read-modify-write). Описание RMW опущено для краткости.

Событие чтения может быть неатомарным, *расслабленным* (relaxed, `rlx`), *потребляющим* (consume, `con`), *приобретающим* (acquire, `acq`) или *последовательно-консистентным* (sequentially-consistent, `sc`). Событие записи может быть неатомарным, расслабленным, *высвобождающим* (release, `rel`) или последовательно-консистентным. В обоих случаях виды обращений упорядочены по величине предоставляемых гарантий. Так, расслабленные обращения похожи

на обычные обращения в модели JMM, тогда как последовательно-консистентные — на синхронизирующие. Неатомарные обращения предоставляют наименьшие гарантии: если неатомарное обращение является частью гонки по данным, то в C/C++11 MM программа имеет неопределенное поведение. Приобретающее чтение и высвобождающая запись могут быть использованы для того, чтобы запретить результат $[a = 1, b = 0]$ (аналогично синхронизирующим обращениям в случае JMM):

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; & \\ [x] :=_{\texttt{rlx}} 1; & a :=_{\texttt{acq}} [y]; \quad //1 \\ [y] :=_{\texttt{rel}} 1 & b :=_{\texttt{rlx}} [x] \quad //0 \end{array}$$

При этом в отличие от синхронизирующих обращений JMM и `sc`-обращений C/C++11 MM на высвобождающих и приобретающих событиях не существует тотального порядка, учитывающего программный порядок, и, как следствие, следующая программа, использующая высвобождающие записи, имеет сценарий поведения с результатом $[a = 1, b = 1]$:

$$\begin{array}{l|l} [x] :=_{\texttt{rel}} 1; & [y] :=_{\texttt{rel}} 1; \\ [y] :=_{\texttt{rel}} 2 & [x] :=_{\texttt{rel}} 2 \\ a :=_{\texttt{sc}} [x]; //1 & \\ b :=_{\texttt{sc}} [y] \; //1 & \end{array} \tag{2W-rel}$$

В то же время эта программа с `sc`-записями такого сценария не имеет:

$$\begin{array}{l|l} [x] :=_{\texttt{sc}} 1; & [y] :=_{\texttt{sc}} 1; \\ [y] :=_{\texttt{sc}} 2 & [x] :=_{\texttt{sc}} 2 \\ a :=_{\texttt{sc}} [x]; //1 & \\ b :=_{\texttt{sc}} [y] \; //1 & \end{array} \tag{2W-sc}$$

Инструкция высвобождающей записи не может быть переставлена инструкциями записями, предшествующими ей в программном порядке, ни на этапе компиляции, ни во время исполнения программ. Как следствие, гарантируется, что поток, который с помощью инструкции приобретающего чтения прочитает из высвобождающей записи, будет осведомлён о предшествующих ей записях. Аналогично, инструкция приобретающего чтения не может быть переставлена с последующими чтениями.

Инструкция потребляющего (consume, `con`) чтения является ослабленной версией инструкции приобретающего чтения и предоставляет те же гарантии, но

только для последующих чтений, которые зависят по данным от потребляющего. Так, потребляющего чтени будет недостаточно, чтобы гарантировать отсутствие сценария поведения $[a = 1, b = 0]$ в версии программы MP-rel-acq, в которой `acq`-чтение заменено на `con`-чтение. Тем не менее, `con`-чтения будет достаточно, чтобы гарантировать отсутствие сценария поведения $[a = y, b = 0]$ в следующей программе:

$$\begin{array}{l|l|l} [x] := 0; [y] := 0; [z] := x; & & \\ [y] :=_{\texttt{rlx}} 1; & a :=_{\texttt{con}} [z]; & //y \\ [z] :=_{\texttt{rel}} y & b :=_{\texttt{rlx}} [a] & //0 \end{array} \qquad \text{(MP-addr-con)}$$

## Сценарии поведения

Аналогично модели JMM, в C/C++11 MM сценарий поведения программы представляется в виде графа, вершинами которого являются события над памятью. Также в сценариях поведения фигурируют отношения программного порядка `po`, “читает из” `rf`, “синхронизируется с” `sw` и “предшествует” `hb`. При этом `sw` и `hb` определяются по-другому. Так, в частности, `sw` включает в себя подмножество `rf`, связывающее высвобождающие записи с приобретающими чтениями, а `hb` является транзитивным замыканием объединения `po` и `sw` в отсутствии `con`-чтений (с ними определение становится сложнее). Дополнительно в модели используются два важных отношения — `mo` и `sc`. Отношение `mo` (*порядок памяти*, memory order, coherence order) для каждой локации является полным порядком на событиях записи в неё. Отношение `sc` (sequential consistency order) является полным порядком на всех `sc`-событиях в сценарии поведения.

Также, как и в случае JMM, для C/C++11 MM определяются аксиомы, которые ограничивают пространство возможных сценариев поведения. Полный список аксиом представлен в [4, 65]. Отличием от JMM является то, что в C/C++11 MM не используется процесс верификации исполнения. В итоге, с одной стороны, C/C++11 MM допускает поведения со “значениями из воздуха”, но, с другой стороны, поддерживает большее число компиляторных оптимизаций.

## Построение сценариев поведения по программе

Построение сценариев поведения в C/C++11 MM состоит из трёх следующих этапов. На первом этапе для программы строится множество т.н. *предзапусков* (preexecutions) — частичных графов сценариев поведения, в которых присут-

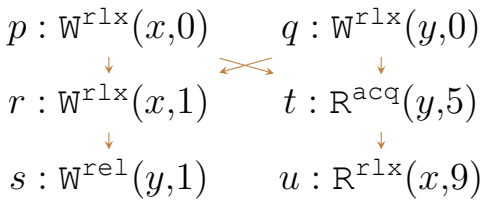



Рис. 2. Предзапуск программы MP-rel-acq

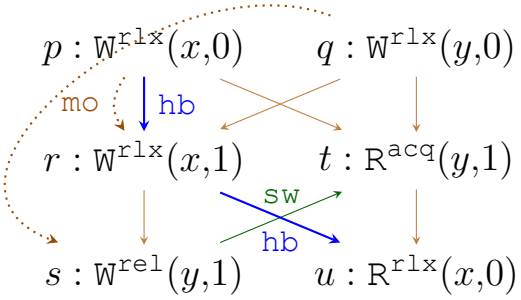



Рис. 3. Сценарий поведения программы MP-rel-acq, прошедший базовую проверку на корректность

ствует только отношение po (аналогичный процесс для аксиоматической модели ARMv8.3 [10] представлен в приложении Г.2). При этом, поскольку события чтения не связываются с событиями записи, для каждого события чтения прочитанное значение выбирается недетерминировано. В частности, среди предзапусков программы MP-rel-acq присутствует граф, изображенный на рис. 2.

На втором этапе в каждый предзапуск недетерминировано добавляются дуги отношений rf, sc и mo, по которым также вычисляются производные отношения, такие как sw и hb. Полученные графы проверяются на базовую корректность. В частности, проверяется, что если из события $e$ идёт rf-дуга в $f$, то $e$ является событием записи, $f$ — чтения, они оперируют над одной и той же локацией, и прочитанное значение равняется записанному. После второго этапа во множестве полученных сценариев поведения для программы MP-rel-acq появляется сценарий поведения, изображенный на рис. 3 (некоторые ребра в нём опущены).

На третьем этапе полученные сценарии поведения проверяются на соответствие аксиомам C/C++11 MM. Одна из таких аксиом аналогична приведённой выше для JMM:

$$\forall r, w.\ \mathtt{rf}(w, r) \wedge \mathtt{hb}(w, r) \Rightarrow \\ \nexists w'.\ \mathtt{mo}(w, w') \wedge \mathtt{hb}(w, w') \wedge \mathtt{hb}(w', r).$$

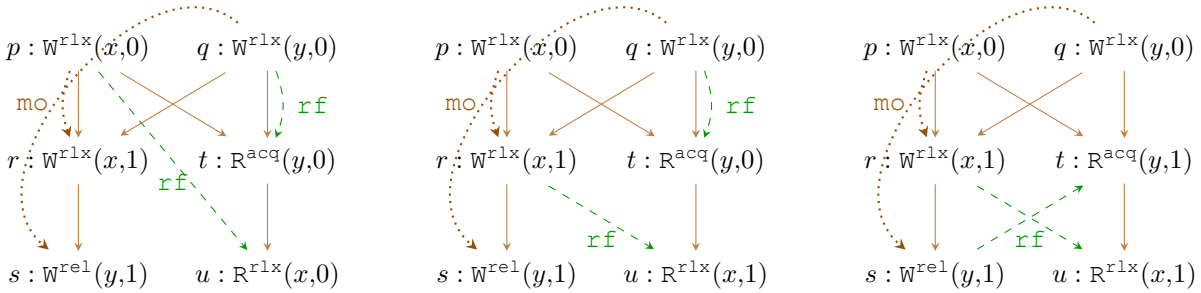



Рис. 4. Сценарии поведения программы MP-rel-acq в C/C++11 MM

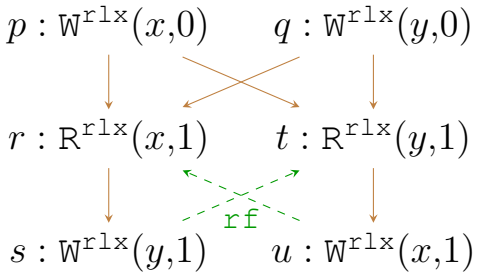



Рис. 5. Сценарий поведения программы IF-OOTA-rlx

Данная аксиома не выполняется для последнего приведенного сценария, и, как следствие, программа MP-rel-acq имеет только три корректных сценария поведения в рамках C/C++11 MM (см. рис. 4), соответствующих результатам $[a = 0, b = 0]$, $[a = 0, b = 1]$ и $[a = 1, b = 1]$.

## Проблема “значений из воздуха”

Для того, чтобы понять, откуда берется проблема “значений из воздуха” в C/C++11 MM, рассмотрим программу, которая уже была обсуждена ранее:

$$
\begin{array}{l||l}
[x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\
a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y]; \\
\textbf{if } a & \textbf{if } b \\
\textbf{then } [y] :=_{\texttt{rlx}} 1 & \textbf{then } [x] :=_{\texttt{rlx}} 1 \\
\textbf{else skip} & \textbf{else skip} \\
\textbf{fi} & \textbf{fi}
\end{array}
\qquad \text{(IF-OOTA-rlx)}
$$

Одним из корректных сценариев поведения этой программы в C/C++ MM является граф, изображённый на рис. 5. Он обладает “значениями из воздуха”, т.к. соответствует результату $[a = 1, b = 1]$. В данном случае очевидно, что у события $r$ есть циклическая зависимость от себя самого — для того, чтобы событие

$r$ прочитало 1, событие $u$ должно записать 1, и это возможно только если событие $t$ прочитало значение 1, которое должно была быть записано событием $s$, которое случается только если $r$ прочитало 1!

Мы могли бы попробовать решить данную проблему, добавив в граф отношение зависимости по управлению `ctrl` с рёбрами $(r, s)$ и $(t, u)$, а также дополнительную аксиому, которая требовала бы ацикличности объединения отношений `ctrl` и `rf`. Тогда в приведенном графе был бы цикл $[r, s, t, u, r]$, и сценарий считался бы некорректным с точки зрения модифицированной модели.

К сожалению, такое решение является неудовлетворительным с точки зрения задач C/C++11 MM. Рассмотрим следующую программу:

| | | |
|---|---|---|
| | $[x] :=_{\texttt{rlx}} 0;$ | $[y] :=_{\texttt{rlx}} 0;$ |
| $a :=_{\texttt{rlx}} [x];$ | $\parallel$ | $b :=_{\texttt{rlx}} [y];$ |
| **if** $a$ | | **if** $b$ |
| **then** $[y] :=_{\texttt{rlx}} 1$ | | **then** $[x] :=_{\texttt{rlx}} 1$ |
| **else** $[y] :=_{\texttt{rlx}} 1$ | | **else skip** |
| **fi** | | **fi** |

(IF-notOOTA-rlx)

Она отличается от IF-OOTA-rlx тем, что ветка **else** в первом потоке содержит инструкцию записи $[y] :=_{\texttt{rlx}} 1$. Наличие этой инструкции позволяет компилятору сделать вывод о том, что инструкция $[y] :=_{\texttt{rlx}} 1$ не зависит от условия конструкции **if**, следовательно может быть вынесена из этой конструкции. Далее вынесенная операция может быть переупорядочена с независимым чтением $a :=_{\texttt{rlx}} [x]$:

| | | |
|---|---|---|
| | $[x] :=_{\texttt{rlx}} 0;$ | $[y] :=_{\texttt{rlx}} 0;$ |
| $[y] :=_{\texttt{rlx}} 1;$ | $\parallel$ | $b :=_{\texttt{rlx}} [y];$ |
| | | **if** $b$ |
| $a :=_{\texttt{rlx}} [x]$ | | **then** $[x] :=_{\texttt{rlx}} 1$ |
| | | **else skip** |
| | | **fi** |

(IF-notOOTA-optimized)

Для получившейся программы результат $[a = 1, b = 1]$ возможен также и в модели SC.

Таким образом для того, чтобы разрешить использованные трансформации, C/C++11 MM должна разрешать результат $[a = 1, b = 1]$ для программы IF-notOOTA-rlx. Это означает, что предложенное ранее “исправление” модели не

может быть напрямую использовано, т.к. добавление зависимостей по управлению в граф и дополнительной аксиомы в модель сделает результат $[a = 1, b = 1]$ некорректным для IF-notOOTA-rlx.

Рассмотрим, в чём заключается разница между зависимостями по управлению в левых потоках программ IF-OOTA-rlx и IF-notOOTA-rlx. В обоих случаях присутствует синтаксическая зависимость инструкции $[y] :=_{\texttt{rlx}} 1$ от $a :=_{\texttt{rlx}} [x]$, но семантическая зависимость имеется только в программе IF-OOTA-rlx. Соответственно, проблема могла бы быть решена, если бы в граф сценария поведения была добавлена семантическая зависимость по управлению [94]. В следующем разделе рассмотрены существующие на данный момент способы представления семантической зависимости.

Стоит отметить, что в стандартах C [30] и C++ [31, 82] упомянутую проблему пытаются решить путём прямого запрета сценариев поведения со “значениями из воздуха”. Данный запрет является неформальным, поскольку в стандарте отсутствует формальное определение “значений из воздуха”. Но в [5] отмечается, что такое решение проблемы является неудовлетворительным, поскольку не даёт возможности строго рассуждать о программах.

### 1.3.4 Теоретические модели памяти

В этом разделе коротко рассматриваются альтернативные теоретические модели памяти для языков программирования.

В [40] определяется операционная семантика на базе C/C++11 MM. Эта семантика по предзапуску программы инкрементально строит полный сценарий поведения. Данный подход решает проблему интеграции C/C++11 MM с остальной частью стандарта [79], однако оставляет проблему “значений из воздуха”, а также не предоставляет операционной интуиции о поведении программы.

В [35] предлагается альтернативный подход к аннотированию инструкций обращений к памяти: вместо того, чтобы помечать часть инструкций “строгими” модификаторами, такими как `acq`, `rel` и `sc`, и значит неявно запрещать переупорядочивание инструкций вокруг них, авторы [35] предлагают явно указывать то, порядок между какими инструкциями должен быть сохранён. Такой подход позволяет эффективнее компилировать программы в целевую платформу. Сама модель памяти в данной работе представлена в операционном виде. К её недостат-

кам стоит отнести то, что в ней позволяются сценарии поведения со “значениями из воздуха”.

В [38] представлена модели памяти на основе *структур событий* (event structures) [95,96]. Семантика программы представляется не как множество несвязанных графов, а как один общий граф, в котором присутствует *отношение конфликта*. Последнее позволяет разделять вершины, относящиеся к разным сценарием поведения — каждый максимальный подграф, в котором нет конфликтующих событий, является одним сценарием поведения в аксиоматическом смысле. Поскольку в структуре событий все сценарии представляются согласованно, данная семантика может проверять наличие семантической зависимости между событиями. В частности, для программы IF-notOOTA-rlx семантика может проверить, что вне зависимости от того, какое значение читается в регистр $a$ левым потоком, он исполняет запись $[y] :=_{\texttt{rlx}} 1$, следовательно между соответствующими чтением и записью нет зависимости. Аналогично, для программы IF-OOTA-rlx семантика может определить наличие зависимости. Данная модель обладает следующими недостатками. Во-первых, она поддерживает только фиксированное множество компиляторных оптимизаций, которые явным образом представлены как переходы в модели. Во-вторых, модель определена для подмножества языка C/C++, содержащего только неатомарные и расслабленные операции чтения и записи, а также захват и высвобождение замков. И, в-третьих, модель не поддерживает эффективную компиляцию в модель ARMv8 POP [9][5].

Близкий подход к заданию модели памяти без “значений из воздуха” представлен в [39]. В этой модели, в отличие от использования явных переписываний структуры событий с помощью фиксированного набора оптимизаций, задан процесс валидации корректных сценариев поведения, похожий на аналогичный механизм в JMM. Процесс валидации основан на идее *игры* двух игроков. Первый игрок пытается обойти подграф структуры событий, соответствующий проверяемому сценарию; второй игрок вмешивается в этот обход, добавляя в конфигурацию обхода произвольные вершины и пытаясь помешать первому. При наличии выигрышной стратегии у первого игрока считается, что сценарий поведения корректен с точки зрения модели. Эта игра является представлением недетерминированного переключения контекстов со стороны процессора. Для модели формально доказано утверждение, косвенно подтверждающие отсутствие проблемы “значений из

[5]В главе 3 рассматривается программа ARM-weak. Эта программа обладает слабым поведением, которое не проявляется в модели [38].

воздуха”. К недостаткам подхода стоит отнести то, что он не поддерживает некоторые базовые оптимизации, в частности, перестановку независимых чтений.

## 1.4 Выводы

На основе проделанного обзора сделаны следующие выводы.

– Модель памяти промышленного языка программирования должна удовлетворять, как минимум, трём критериям. Во-первых, должна существовать корректная схема компиляции в модель целевой процессорной архитектуры. Во-вторых, основные компиляторные оптимизации должны быть корректны в рамках модели. В-третьих, у модели должна отсутствовать проблема “значений из воздуха”.
– При разработке новой модели памяти языка программирования нужно доказывать корректность эффективной компиляции в модели памяти целевых процессорных архитектур.
– Существующие модели памяти промышленных языков программирования не удовлетворяют приведённым выше критериям.
– Требуется разработать операционную модель памяти с синтаксисом C/C++11 MM, которая не имеет проблемы “значений из воздуха”.

# Глава 2. Операционная модель памяти C/C++11

В данной главе описана предложенная в диссертации операционная модель памяти OpC11 для языков C/C++11, и представлен реализующий её интерпретатор. Материал главы покрывает результат статьи [25].

Модель OpC11 (далее OpC11 MM) представлена как семейство аспектов, каждый из которых описывает некоторую особенность C/C++11 MM [4]. Это позволяет упростить использование модели для задач, в которых рассматривается только подмножество языка C/C++11 MM. Также, поскольку некоторые особенности C/C++11 MM, такие как поддержка секвенциализации (см. раздел 2.1.8), неоднозначны, то аспектное представление позволяет легко настраивать интерпретатор на интересующий вариант семантики.

Описание модели организовано следующим образом. В начале рассматриваются ряд примеров, мотивирующих введение различных аспектов OpC11 MM, и неформально описываются сами аспекты (раздел 2.1). Далее приводится математическое определение упомянутых аспектов (раздел 2.2). Обсуждаются результаты тестирования модели на наборе “лакмусовых” тестов (litmus tests) и RCU-структуре (раздел 2.3). В заключении главы обсуждаются свойства модели (раздел 2.4).

## 2.1 Основные понятия

OpC11 MM задается операционным способом, следовательно существует абстрактная машина, связанная с моделью, которая исполняет программы по шагам. Далее для обозначения этой модели используется термин *машина* OpC11.

### 2.1.1 Память и базовый фронт

Состояние машины OpC11 включается в себя множество сообщений, которое называется *памятью*. *Сообщение* — это тройка, состоящая из локации, значения и *метки времени* (timestamp). Метка времени является натуральным числом и используется для упорядочивания сообщений, которые связаны с одной и той же локацией. Такой порядок аналогичен отношению `mo` в аксиоматической C/C++11 MM [4].

Рассмотрим машину OpC11 на примере исполнения следующей программы:

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; & \\ [x] :=_{\texttt{rlx}} 1; & a :=_{\texttt{rlx}} [y]; \\ [y] :=_{\texttt{rlx}} 1 & b :=_{\texttt{rlx}} [x]; \\ & c :=_{\texttt{rlx}} [x] \end{array} \qquad \text{(MP-rlx-2)}$$

Данная программа является очередной вариацией программы MP, в которой во второй поток было добавлено дополнительное чтение из локации $x$. После исполнения первых двух инструкций ($[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0$) память машины будет содержать два сообщения:

$$M = \{\langle x : 0@\mathbf{0}\rangle, \langle y : 0@\mathbf{0}\rangle\},$$

где **0** – это метка времени. После того, как левый поток закончит своё исполнение, в памяти будет находится четыре сообщения:

$$M = \{\langle x : 0@\mathbf{0}\rangle, \langle y : 0@\mathbf{0}\rangle, \langle y : 1@\mathbf{1}\rangle, \langle x : 1@\mathbf{1}\rangle\}.$$

Заметим, что в C/C++11 MM [4] у рассматриваемой программы есть сценарий поведения с результатом $[a = 1, b = 0, c = 0]$. Для того, чтобы разрешить этот результат, потоки машины OpC11 имеют право при чтении выбирать из памяти сообщение не с самой большой меткой времени, как этого требовала бы абстрактная машина, представляющая модель SC, т.е. не самую последнюю запись в локацию. После завершения исполнения левого потока правый поток может сначала прочитать сообщение $\langle y : 1@\mathbf{1}\rangle$, присвоив в регистр $a$ значение 1, а затем выполнить два чтения из старого сообщения $\langle x : 0@\mathbf{0}\rangle$, получив $b = c = 0$.

Отметим, что C/C++11 MM запрещает сценарий поведения с результатом $[a = 1, b = 1, c = 0]$, поскольку гарантирует, что если поток “увидел” более новую запись в локацию (в данном случае это сообщение $\langle x : 1@\mathbf{1}\rangle$), то он не может прочитать более старое, с точки зрения отношения $\texttt{mo}$, сообщение. Для того, чтобы поддержать данное ограничение, в OpC11 MM у каждого потока есть т.н. *базовый фронт*, $\mathit{view}_{\text{cur}}$. Базовый фронт — это частичная функция, которая по локации возвращает максимальную метку времени сообщения, связанного с локацией, о котором осведомлен поток. Когда поток пытается прочитать или записать сообщение в локацию $\ell$ с меткой времени $\tau$, он проверяет, что $\tau$ больше или равно, чем значение его базового фронта по $\ell$[1].

[1] Строго говоря, поток никогда не “пытается” делать какое-то действие. В формальной модели упомянутая проверка реализуется как дополнительное условие в правилах чтения и записи.

Рассмотрим действие базового фронта на примере сценария поведения программы MP-rlx-2, который приводит к результату $[a = 1, b = 1, c = \_]$. Изначально в системе существует один поток $T0$ с пустым базовым фронтом. После исполнения первых двух инструкций программы ($[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0$) базовый фронт потока $T0.\mathit{view}_{\text{cur}}$ будет указывать на соответствующие сообщения из памяти:

$$T0.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

Далее машина стартует два потока $T1$ и $T2$, базовые фронты которых будут равны $T0.\mathit{view}_{\text{cur}}$. После исполнения инструкции (левого) потока $T1$ память машины будет содержать два новых сообщения, а базовый фронт потока $T1$ станет равным **1** по обоим локациям, т.к. поток, сделавший запись, естественным образом осведомлён о ней:

$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После этого (правый) поток $T2$ может прочитать новое сообщение в локацию $y$, присвоив 1 в регистр $a$ и увеличив свой фронт по локации $y$ до **1**:

$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{1}].$$

Т.к. базовый фронт потока $T2$ по локации $x$ равен **0**, т.е. поток ещё не осведомлён о новой записи в $x$, то поток может прочитать либо сообщение $\langle x : 0@\mathbf{0}\rangle$, либо сообщение $\langle x : 1@\mathbf{1}\rangle$. Для того, чтобы $b$ равнялось 1, поток $T2$ должен выполнить чтение из более нового сообщения, что обновит его базовый фронт по локации $x$:

$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}].$$

После этого потоку $T2$ остаётся только выполнить последнее чтение ($c :=_{\texttt{rlx}} [x]$), и поскольку его базовый фронт по локации $x$ равен **1**, то он может прочитать только из нового сообщения в локацию $x$ ($\langle x : 1@\mathbf{1}\rangle$). Как следствие, результат $[a = 1, b = 1, c = 0]$ невозможен для программы MP-rlx-2 в OpC11 MM.

### 2.1.2 Синхронизация потоков

Рассмотрим программу MP-rel-acq из главы 1.

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; & \\ [x] :=_{\texttt{rlx}} 1; & a :=_{\texttt{acq}} [y]; \\ [y] :=_{\texttt{rel}} 1 & b :=_{\texttt{rlx}} [x] \end{array} \qquad \text{(MP-rel-acq)}$$

Для этой программы результат $[a = 1, b = 0]$ запрещён в C/C++11 MM, т.к. если $a = 1$, то между потоками произошла синхронизация (т.е. в соответствующем графе есть ребро отношения `sw`), и перед выполнением $b :=_{\texttt{rlx}} [x]$ правый поток должен быть осведомлен о записи $[x] :=_{\texttt{rlx}} 1$.

Для того, чтобы выразить такую синхронизацию, у каждого сообщения машины OpC11 есть четвертая дополнительная компонента — *фронт сообщения*. Фронт сообщения $m$ хранит информацию о сообщениях, о которых узнаёт поток, выполнивший приобретающее (`acq`) чтение сообщения $m$. Если поток $T$ выполняет высвобождающую (`rel`) запись, то фронтом сообщения, которое будет добавлено в память как результат исполнения записи, будет базовый фронт потока $T$ на момент выполнения записи. Расслабленные (`rlx`) записи также помещают фронт в сообщения, но в соответствии с более сложными правилами, которые будут описаны в разделе 2.1.9.

Рассмотрим сценарий поведения MP-rel-acq, в результате которого $a = 1$. После того, как выполнены две инициализирующие записи и запущены два потока, память и базовые фронты потоков выглядят следующим образом:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После исполнения двух записей (левым) потоком $T1$ в память попадает два новых сообщения, одно из которых было сделано высвобождающей записью:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После того, как (правый) поток выполняет приобретающее чтение из сообщения $\langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle$, его базовый фронт увеличивается по обоим компонентам:

$$T2.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}].$$

После этого (правый) поток $T2$ не может прочитать старое сообщение $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$. Таким образом OpC11 MM запрещает результат $[a = 1, b = 0]$ для программы MP-rel-acq.

### 2.1.3 Операционные буфера

Тем не менее, не все слабые сценарии поведения программ, наблюдаемые в C/C++11 MM, могут быть описаны приведенными выше способами. Рассмотрим

программу LB-rlx (load buffering, буферизация записи):

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y]; \\ [y] :=_{\texttt{rlx}} 1 & [x] :=_{\texttt{rlx}} 1 \end{array} \qquad \text{(LB-rlx)}$$

C/C++11 MM допускает сценарий поведения этой программы с результатом $[a = 1, b = 1]$. Такой результат требует, чтобы в момент исполнения инструкции $a :=_{\texttt{rlx}} [x]$ в памяти находилось сообщение $\langle x : 1@\_,\_\rangle$. Это означает, что инструкция $[x] :=_{\texttt{rlx}} 1$ должна быть выполненной. Аналогичное утверждение верно и для пары инструкций $b :=_{\texttt{rlx}} [y]$ и $[y] :=_{\texttt{rlx}} 1$. Таким образом, хотя бы в одном из потоков инструкция записи должна быть исполнена раньше чтения.

Для решения этой проблемы OpC11 MM добавляет в состояние каждого потока по *операционному буферу* — списку записей об отложенных инструкциях, которые хранят всю необходимую информацию для дальнейшего исполнения. Так, в частности, когда поток откладывает инструкцию чтения, он заменяет её в программе на новое, уникальное символьное значение[2], а в буфер добавляет пару, состоящую из символьного значения и целевой локации. Для отложенной инструкции записи в буфер сохраняется целевая локация и значение, которое нужно записать. Далее поток машины OpC11 может недетерминировано выбрать отложенную инструкцию из буфера и исполнить её, если в буфере перед ней нет инструкции, которая может непосредственно повлиять на результат выбранной. Так, данный механизм позволяет в программе LB-rlx отложить исполнение инструкции чтения в левом потоке, что даёт возможность получить результат $[a = 1, b = 1]$.

### 2.1.4 Спекулятивное исполнение

C/C++11 MM поддерживает оптимизацию, которая выносит за пределы условного оператора инструкцию, которая встречается в обоих ветках оператора. Рассмотрим программу SE-simple (speculative execution, спекулятивное исполне-

[2]OpC11 MM описана в стиле редукционных контекстов [13, 18], т.е. программа представляется как выражение, которое постепенно редуцируется. Как следствие, инструкция чтения в этой семантике — это некоторое подвыражение, которое, будучи вычисленным, заменяется на прочитанное значение. Подробное описание семантики в виде редукционных контекстов приведено в разделе 2.2.

ние):

$$
\begin{array}{l}
[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; [z] :=_{\texttt{rlx}} 0; \\
\begin{array}{l|l}
a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y]; \\
\textbf{if } a & \textbf{if } b \\
\textbf{then } [z] :=_{\texttt{rlx}} 1; & \textbf{then } [x] :=_{\texttt{rlx}} 1 \\
\quad\quad [y] :=_{\texttt{rlx}} 1 & \textbf{else skip} \\
\textbf{else } [y] :=_{\texttt{rlx}} 1 & \textbf{fi} \\
\textbf{fi} &
\end{array} \\
c :=_{\texttt{rlx}} [z]
\end{array}
\qquad \text{(SE-simple)}
$$

В этой программе инструкция $[y] :=_{\texttt{rlx}} 1$ не зависит от условия **if**-оператора, и поэтому может быть выполнена перед инструкцией $a :=_{\texttt{rlx}} [x]$. Это позволяет получить результат $c = 1$.

Для того, чтобы поддержать такие сценарии, в OpC11 MM операционные буфера могут быть *вложенными*. Когда исполнение потока подходит к условному оператору, условие которого зависит от ранее отложенной операции, модель добавляет в буфер кортеж, который содержит символическое представление условия, а также два пустых буфера. Эти буфера в дальнейшем будут пополняться отложенными инструкциями **then** и **else** веток оператора.

Рассмотрим сценарий поведения программы SE-simple, в результате которого $c = 1$. После исполнения инициализирующих инструкций записи память машины OpC11 содержит три сообщения:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle z : 0@\mathbf{0}, [z@\mathbf{0}]\rangle\}.$$

Далее левый поток откладывает выполнение инструкции чтения и начинает спекулятивно, т.е. без вычисления значения условия, исполнять инструкции веток условного оператора. Так, в буфере левого потока оказывается две записи:

$$\langle a :=_{\texttt{rlx}} [x]; \textbf{if } a \ \langle\rangle \ \langle\rangle\rangle.$$

Продолжая (спекулятивно) откладывать инструкции, левый поток помещает все инструкции из условного оператора в соответствующие подбуфера:

$$\langle a :=_{\texttt{rlx}} [x]; \textbf{if } a \ \langle [z] :=_{\texttt{rlx}} 1; [y] :=_{\texttt{rlx}} 1\rangle \ \langle [y] :=_{\texttt{rlx}} 1\rangle\rangle.$$

После того, как в подбуферах появляются одинаковые инструкции (в данном случае это $[y] :=_{\texttt{rlx}} 1$), и перед ними в подбуферах нет конфликтующих инструкций,

OpC11 MM может вынести их на предыдущей уровень буфера:

$$\langle a \ :=_{\texttt{rlx}} \ [x]; [y] \ :=_{\texttt{rlx}} \ 1; \textbf{if} \ a \ \langle [z] \ :=_{\texttt{rlx}} \ 1 \rangle \ \langle\rangle \rangle.$$

Далее, поскольку инструкции $a \ :=_{\texttt{rlx}} \ [x]$ и $[y] \ :=_{\texttt{rlx}} \ 1$ независимы, то $[y] \ :=_{\texttt{rlx}} \ 1$ может быть выполнена, после чего в памяти появляется новое сообщение:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle z : 0@\mathbf{0}, [z@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [y@\mathbf{1}]\rangle\}.$$

После этого результат $c = 1$ получается следующим образом. Правый поток читает из нового сообщения, присваивая 1 в $b$. Поскольку в правом потоке условие после этого было вычислено к значению 1, то может быть выполнена инструкция записи в локацию $x$:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle z : 0@\mathbf{0}, [z@\mathbf{0}]\rangle, \\ \langle y : 1@\mathbf{1}, [y@\mathbf{1}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}]\rangle\}.$$

После этого левый поток может выполнить отложенное чтение $a \ :=_{\texttt{rlx}} \ [x]$ из добавленного сообщения. Так, в буфере левого потока остаётся только одна запись, соответствующая отложенной конструкции **if**, чьё условие было вычислено к значению 1:

$$\langle \textbf{if} \ 1 \ \langle [z] \ :=_{\texttt{rlx}} \ 1 \rangle \ \langle\rangle \rangle.$$

После вычисления отложенной конструкции **if** и выполнения записи в $z$ в памяти находится шесть сообщений:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle z : 0@\mathbf{0}, [z@\mathbf{0}]\rangle, \\ \langle y : 1@\mathbf{1}, [y@\mathbf{1}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}]\rangle, \langle z : 1@\mathbf{1}, [z@\mathbf{1}]\rangle\}.$$

Чтение из последней записи в локацию $z$ приводит к результату $c = 1$.

Идея перевода повторяющихся инструкций из вложенных буферов естественным образом обобщается на случай вложенных условных операторов.

### 2.1.5 `sc`-инструкции

C/C++11 MM гарантирует наличие тотального порядка `sc` на `sc`-событиях, который не противоречит программному порядку `po`, отношению “синхронизируется с” `sw`, и порядку памяти `mo`. Помимо этого `sc`-чтение обладает теми же свойствами, что и приобретающее чтение, а `sc`-записи — что и высвобождающие записи.

Рассмотрим программу SB-sc (store buffering, буферизация записи):

$$\begin{array}{l|l} [x] :=_{\texttt{sc}} 0; & [y] :=_{\texttt{sc}} 0; \\ [x] :=_{\texttt{sc}} 1; & [y] :=_{\texttt{sc}} 1 \\ a :=_{\texttt{sc}} [y]; & b :=_{\texttt{sc}} [x]; \end{array} \qquad \text{(SB-sc)}$$

Для этой программы C/C++11 MM запрещает результат $[a = 0, b = 0]$. Это объясняется тем, что в её сценарии поведения, как минимум, одно из чтений находится позже все остальных событий в отношении $\texttt{sc}$, а это значит, что соответствующий поток в момент исполнения этого чтения осведомлён о всех ($\texttt{sc}$-)записях в программе.

Для того, чтобы гарантировать аналогичное ограничение, в состояние машины OpC11 добавляется глобальный (общий для всех потоков) компонент — $\texttt{sc}$-*фронт view*$_{\text{sc}}$. Он используется следующим образом: при выполнении $\texttt{sc}$-чтения из локации $\ell$ поток обновляет свой базовый фронт на $[\ell@\mathit{view}_{\text{sc}}(\ell)]$, а при выполнении $\texttt{sc}$-записи в локацию $\ell$ сообщения с меткой времени $\tau$ поток обновляет $\texttt{sc}$-фронт на $[\ell@\tau]$. Таким образом, $\texttt{sc}$-чтение из локации $\ell$ не может прочитать сообщение в памяти, которое имеет меньшую метку времени, чем метка сообщения в ту же локацию, которое было записано последней на тот момент $\texttt{sc}$-записью.

### 2.1.6 Неатомарные обращения

Согласно C/C++11 MM, программы с гонками по данным [97], в которых участвуют неатомарные обращения к памяти, обладают неопределенном поведением. Так, рассмотрим программу DR-rlx-na (data race, гонка по данным):

$$\begin{array}{l|l} [d] :=_{\texttt{na}} 0; & \\ [d] :=_{\texttt{rlx}} 1; & a :=_{\texttt{na}} [d]; \end{array} \qquad \text{(DR-rlx-na)}$$

В ней есть гонка по данным между инструкциями $[d] :=_{\texttt{rlx}} 1$ и $a :=_{\texttt{na}} [d]$. Мы можем идентифицировать эту гонку с помощью базовых фронтов в случае сценария, в котором сначала левый поток выполняет запись, а потом правый — чтение. В такой ситуации правый поток будет выполнять неатомарную операцию над локацией $d$, при этом не являясь осведомленным о последний записи в данную локацию. Если OpC11 MM идентифицировала гонку по данным, в которую вовлечена неатомарная операция хотя бы в одном сценарии поведения программы, то считается, что программа в целом обладает неопределенным поведением.

Рассмотрим похожую программу, также содержащую гонку по данным:

$$\begin{array}{c} [d] :=_{\texttt{na}} 0; \\ [d] :=_{\texttt{na}} 1; \;\|\; a :=_{\texttt{rlx}} [d]; \end{array} \qquad \text{(DR-na-rlx)}$$

Идентификация гонки в данном случае не может быть проведена только с помощью базовых фронтов потоков. Для решения проблемы в состояние машины OpC11 вводится ещё один глобальный фронт — `na`-*фронт* $\mathit{view}_{\text{na}}$. Подобно `sc`-фронту, `na`-фронт по локации возвращает метку времени последней `na`-записи в неё. Так, при выполнении любой операции над локацией $\ell$ поток проверяет, что он осведомлён о последней `na`-записи в $\ell$ (т.е. его базовый фронт по $\ell$ больше или равен $\mathit{view}_{\text{na}}(\ell)$), и если это не так, то машина OpC11 сигнализирует о гонке.

С помощью такой техники гонка по данным в программе DR-na-rlx идентифицируется в сценарии поведения, в котором `na`-запись левого потока выполняется до `rlx`-чтения правого потока.

### 2.1.7 Потребляющие чтения

Потребляющее (consume, `con`) чтение является более слабой версией приобретающего чтения. Приобретающее чтение обновляет базовый фронт потока с помощью фронта прочитанного сообщения и тем самым влияет на все последующие инструкции, тогда как потребляющее чтение действует только на последующие чтения, которые зависят от него по адресу[3].

Рассмотрим программу, в которой используется потребляющее чтение:

$$\begin{array}{rcl} [p] :=_{\texttt{na}} \textbf{null}; [d] :=_{\texttt{na}} 0; [x] :=_{\texttt{na}} 0; & & \\ & \| & a :=_{\texttt{con}} [p]; \\ & & \textbf{if } a \neq \textbf{null} \\ [x] :=_{\texttt{rlx}} 1; & & \textbf{then } b :=_{\texttt{na}} [a]; \\ [d] :=_{\texttt{na}} 1; & & \quad\; c :=_{\texttt{rlx}} [x] \\ [p] :=_{\texttt{rel}} d & & \textbf{else } b := 0; \\ & & \quad\; c := 0 \\ & & \textbf{fi} \end{array} \qquad \text{(MP-con-na-2)}$$

В этой программе левый поток передаёт информацию, записанную в локацию $d$, правому потоку через указатель $p$. Правый поток с помощью `con`-чтения заносит

[3] Инструкция чтения $i$ *зависит по адресу* от инструкции чтения $j$, если целевой адрес $i$ зависит от результата исполнения $j$.

в переменную $a$ содержимое указателя $p$. Если переменная $a$ не равна **`null`**, то поток читает из той локации, на которую указывает $a$ (в данном случае на локацию $d$), а потом — из локации $x$. В C/C++11 MM у этой программы существует три возможных результата исполнения: $[a = \texttt{null}, b = 0, c = 0]$, $[a = d, b = 1, c = 1]$ и $[a = d, b = 1, c = 0]$. Если в программе `con`-чтение заменить на `acq`-чтение, то последний результат станет невозможным, т.к. после выполнения $a :=_{\texttt{acq}} [p]$ с результатом $d$ базовый фронт правого потока будет указывать на сообщения, которые были получены в результате исполнения $[x] :=_{\texttt{rlx}} 1$ и $[d] :=_{\texttt{na}} 1$. В то же время `con`-чтение $a :=_{\texttt{con}} [p]$ "синхронизирует" только последующее чтение $b :=_{\texttt{na}} [a]$, т.к. оно является разыменованием результата `con`-чтения и не влияет на независимое чтение $c :=_{\texttt{rlx}} [x]$.

Для поддержки подобного поведения OpC11 MM может *аннотировать* с помощью фронтов инструкции чтения, которые зависят от `con`-чтений. Когда потребляющее чтение получает из памяти сообщение $\langle \ell : v@\tau, \mathit{view} \rangle$, то, вместо того, чтобы обновить базовый фронт потока с помощью *view*, как это сделало бы приобретающее чтение, оно помечает все зависимые от него инструкции чтения фронтом *view*. В дальнейшем, когда помеченная инструкция чтения будет исполняться, она скомбинирует базовый на тот момент фронт потока с фронтом-пометкой для того, чтобы вычислить минимальную метку времени, сообщение с которой доступно для чтения. Аналогичным образом помечаются и обрабатываются отложенные чтения в операционных буферах, которые ссылаются на символьный результат `con`-чтения.

### 2.1.8 Соединение потоков

В момент соединения потоков, т.е. когда оба потока закончили своё исполнение, естественно ожидать, что все отложенные операции были выполнены, и, соответственно, операционные буфера пусты. Это подтверждается тем, что в C/C++11 MM [4] используется отношение "дополнительно синхронизируется с" (additional-synchronizes-with, `asw`), которое связывает последние событие потока с первой следующей за потоком инструкцией родительского потока. Отношение `asw` является частью отношения `hb`. Это означает, что родительский поток оказывается осведомлен о всех сообщениях, которые были прочитаны дочерними потоками. В OpC11 MM это выражается так: соединение потоков возможно, только если их операционные буфера пусты, а после соединения потоков родительский

поток получает базовый фронт, равный комбинации базовых фронтов соединяемых потоков.

Тем не менее, это противоречит стандартам C и C++11, требующим, чтобы оптимизация секвенциализации ($C_1 \mid\mid C_2 \rightsquigarrow C_1;\ C_2$) была корректной. Это делает предыдущие предположения некорректными. Данная проблема может быть проиллюстрирована на следующей программе, в которой параллельная композиция с пустым потоком может быть заменена на не пустой поток:

$$\begin{array}{ccc} & [x]\ :=_{\texttt{rlx}}\ 0; & [y]\ :=_{\texttt{rlx}}\ 0; \\ a\ :=_{\texttt{rlx}}\ [y] & \Big\| \ \texttt{skip} & \Big\| \ b\ :=_{\texttt{rlx}}\ [x] \\ [x]\ :=_{\texttt{rlx}}\ 1 & & [y]\ :=_{\texttt{rlx}}\ 1 \end{array} \qquad \text{(LB-rlx-join)}$$

Если применить такую оптимизацию к одной из композиций потоков, то результат $[a = 1, b = 1]$ станет возможным. Для того, чтобы поддержать данный сценарий поведения, в интерпретаторе модели допускается альтернативное правило соединения потоков как отдельный аспект.

### 2.1.9 Расслабленные обращения и синхронизация

Между расслабленными операциями, высвобождающими записями и приобретающими чтениями существует тонкое взаимодействие. В этом подразделе описываются варианты этого взаимодействия, а также технические решения в OpC11 MM, обеспечивающие его.

#### Высвобождающие цепочки

В связи с тем, что расслабленное чтение не может участвовать в синхронизации, его исполнение не обновляет базовый фронт потока фронтом прочитанного сообщения, как это происходит в случае приобретающего чтения. Тем не менее, если приобретающее чтение потока $T_2$ читает из сообщения, которое было добавлено расслабленной записью $w_{\texttt{rlx}}$ потока $T_1$, то оно должно синхронизироваться с высвобождающей записью в ту же локацию $w_{\texttt{rel}}$, предшествующую $w_{\texttt{rlx}}$ в потоке $T_1$, если такая запись существует. Это свойство является аналогом высвобождающих цепочек (release sequences) из C/C++11 MM [4].

Рассмотрим вариант программы MP, в сценариях поведения которого имеется описанная выше синхронизация:

$$\begin{array}{l} [f] :=_{\texttt{na}} 0; [d] :=_{\texttt{na}} 0; [x] :=_{\texttt{na}} 0; \\ \begin{array}{l} [d] :=_{\texttt{na}} 5; \\ [f] :=_{\texttt{rel}} 1; \\ [x] :=_{\texttt{rel}} 1; \\ [f] :=_{\texttt{rlx}} 2 \end{array} \Bigg\| \begin{array}{l} \textbf{repeat}\ c :=_{\texttt{acq}} [f]; c == 2\ \textbf{end}; \\ a :=_{\texttt{na}} [d]; \\ b :=_{\texttt{rlx}} [x] \end{array} \end{array} \qquad \text{(MP-rel-acq-na-rlx-2)}$$

Единственным значением, которое может получить регистр $a$, является $5$. Это объясняется тем, что последнее приобретающее чтение $c :=_{\texttt{acq}} [f]$ в цикле должно прочитать значение $2$, следовательно синхронизироваться с высвобождающей записью $[f] :=_{\texttt{rel}} 1$. В то же время регистр $b$ может получить как значение $0$, так и $1$, поскольку запись $[f] :=_{\texttt{rel}} 1$, с которой синхронизируется приобретающее чтение, предшествует записи $[x] :=_{\texttt{rel}} 1$, и поэтому информация об этой записи в $x$ не попадает во фронт сообщения, записанного инструкцией $[f] :=_{\texttt{rel}} 1$.

Для того, чтобы выразить такую синхронизацию, у каждого потока в машине OpC11 имеется *фронт записи* (write-front) $\mathit{view}_{\text{write}}$. Этот фронт для каждой локации возвращает метку времени последней высвобождающей записи, сделанной потоком в эту локацию. Когда поток выполняет расслабленную запись в локацию $\ell$ с меткой времени $\tau$, в соответствующее сообщение записывается фронт, который является комбинацией $[\ell@\tau]$ и фронта сообщения в $\ell$ с меткой времени $\mathit{view}_{\text{write}}(\tau)$, которое было записано высвобождающей записью того же потока.

## Отложенные расслабленные обращения и синхронизация

Рассмотрим вариант программы LB, в которой инструкции чтения являются расслабленными, а записи — высвобождающими:

$$\begin{array}{l} [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; \\ \begin{array}{l} a :=_{\texttt{rlx}} [x]; \\ [y] :=_{\texttt{rel}} 1 \end{array} \Bigg\| \begin{array}{l} b :=_{\texttt{rlx}} [y]; \\ [x] :=_{\texttt{rel}} 1 \end{array} \end{array} \qquad \text{(LB-rel-rlx)}$$

Согласно C/C++11 MM эта программа имеет сценарий поведения с результатом $[a = 1, b = 1]$, поскольку высвобождающие записи не накладывают никаких дополнительных ограничений в отсутствии приобретающих чтений. Поэтому OpC11 MM разрешает исполнять высвобождающие записи даже в тот момент, когда предшествующие отложенные инструкции чтения ещё не выполнены.

Тем не менее, требуются ввести некоторые ограничения на то, как инструкции чтения могут быть отложены через высвобождающие записи. Рассмотрим ещё один вариант программы LB.

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{acq}} [x]; & b :=_{\texttt{rlx}} [y]; \\ [y] :=_{\texttt{rlx}} 1 & [x] :=_{\texttt{rel}} 1 \end{array} \qquad \text{(LB-rel-acq-rlx)}$$

В этой программе в левом потоке есть приобретающее чтение, а в правом — высвобождающая запись. Если это чтение прочитает сообщение, сделанное инструкцией $[x] :=_{\texttt{rel}} 1$, т.е. регистр $a$ получит значение 1, то в терминах C/C++11 MM между $b :=_{\texttt{rlx}} [y]$ и $[y] :=_{\texttt{rlx}} 1$ возникнет ребро отношения “предшествует” hb. Следовательно, регистр $b$ не может получить значение 1, т.к. чтение не может прочитать то, что записано последующей за ним записью. Таким образом, OpC11 MM должна предотвращать ситуацию, когда чтение, которое было отложено через высвобождающую запись $W$, выполняется после того, как другой поток прочитал сообщение $W$ с помощью приобретающего чтения.

Для реализации данного ограничения в состояние машины OpC11 добавляется глобальный компонент $\gamma$, который является списком троек, состоящих из локации, метки времени некоторого существующего сообщения и символического значения, идентифицирующего отложенное чтение. При выполнении высвобождающей записи $W$, которая добавляет сообщение локации $\ell$ с меткой времени $\tau$, для каждого отложенного потоком чтения в список $\gamma$ добавляется по тройке $\langle \ell, \tau, \mathsf{x} \rangle$, где $\mathsf{x}$ — символическим значение чтения. При этом приобретающее чтение из сообщения локации $\ell$ с меткой времени $\tau$ возможно только в том случае, если в $\gamma$ не осталось записей вида $\langle \ell, \tau, \_ \rangle$, а выполнение отложенного чтения с символическим значением $\mathsf{x}$ удаляет из $\gamma$ все тройки $\langle \_, \_, \mathsf{x} \rangle$.

Следующая программа иллюстрирует ещё одну особенность, которая связана с откладыванием расслабленных записей через высвобождающие:

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \\ [y] :=_{\texttt{rel}} 1 & [x] :=_{\texttt{rel}} 1 \\ a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y] \end{array} \qquad \text{(WR-rlx-rel)}$$

Согласно C/C++11 MM, результат $[a = 1, b = 1]$ возможен для этой программы, и это требует, чтобы сообщения расслабленных записей попали в память после

сообщений высвобождающих. При этом, если другой (третий) поток выполнит приобретающее чтение из сообщения, сделанного одной из высвобождающих записей, то он должен стать осведомлённым о предшествующей расслабленной записи, а для этого соответствующее сообщение должно находиться в памяти.

Аналогично предыдущему пункту, данная проблема решается с помощью списка $\gamma$. При исполнении высвобождающей записи также, как и для отложенных чтений, в списке $\gamma$ появляются тройки, связанные с отложенными записями. Единственным отличием от обработки отложенных чтений является то, что при выполнении отложенной записи $W$ не только удаляются все тройки $\langle \ell, \tau, \mathsf{x} \rangle$, где $\mathsf{x}$ — символьный идентификатор записи, но и увеличивается фронт сообщения локации $\ell$ с меткой времени $\tau$ на $[\ell' @ \tau']$, где $\ell'$ и $\tau'$ связаны с сообщением $W$.

## 2.2 Формальное описание модели

В разделе приводится математическое определение OpC11 MM, включая определение синтаксиса языка, для которого задана модель, представления памяти и фронтов.

### 2.2.1 Синтаксис языка модели и базовые правила редукции

Синтаксис языка модели представлен на рис. 6. Мета-переменная $\mathsf{e}$ представляет выражения, которые могут быть целыми числами $z$, локациями $\ell$, неизменяемыми локальными переменными $\mathsf{x}$, парами, проекциями пар или бинарными выражениями. Конструкция **`choice`** является недетерминированным выбором между двумя выражениями.

Программы представляются как операторы $\mathsf{s}$, где $\mathsf{x} := \mathsf{s}_1;\ \mathsf{s}_2$ является let-выражением, **`spw`** $\mathsf{s}_1\ \mathsf{s}_2$ — параллельной композицией потоков, $[\iota]_{\mathsf{RM}}$ — выражением чтения из памяти. В примерах также используются дополнительные обозначения:

- оператор **`skip`** вместо константного выражения $0$,
- выражения $\mathsf{x} :=_{\mathsf{RM}} [\iota]$ и $\mathsf{x} := \mathsf{e}$ для обозначения $\mathsf{x} := [\iota]_{\mathsf{RM}};\ \mathsf{x}$ и $\mathsf{x} := \mathsf{e};\ \mathsf{x}$,
- оператор $\mathsf{s}_1; \mathsf{s}_2$ вместо $\mathsf{x}' := \mathsf{s}_1;\ \mathsf{s}_2$, где $\mathsf{x}'$ не встречается в $\mathsf{s}_2$,
- конструкция $\mathsf{s}_1 || \mathsf{s}_2$ для обозначения **`spw`** $\mathsf{s}_1\ \mathsf{s}_2$.

Результатом программы является либо значение $v$, либо значение времени исполнения (run-time value) **`stuck`**, которое символизирует, что что-то пошло не

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{x} \mid z\ (\in \mathbb{Z}) \mid \mathsf{e}_1\ \mathrm{op}\ \mathsf{e}_2 \mid \mathbf{choice}\ \ \mathsf{e}_1\ \mathsf{e}_2 \\
 & & \mathbf{fst}\ \mathsf{e} \mid \mathbf{snd}\ \mathsf{e} \mid (\mathsf{e}_1, \mathsf{e}_2) \mid \iota \\
\mathrm{op} & ::= & + \mid - \mid * \mid / \mid \% \mid == \mid \neq \\
\iota & ::= & \ell \mid \mathsf{x} \\
\ell & — & \text{идентификатор локации} \\
\mathsf{x} & — & \text{локальная переменная} \\
v & ::= & \ell \mid z \mid (v_1, v_2) \\
 & & \\
\mathsf{s} & ::= & \mathsf{e} \mid \mathsf{x} := \mathsf{s}_1;\ \mathsf{s}_2 \mid \mathbf{spw}\ \mathsf{s}_1\ \mathsf{s}_2 \mid \\
 & & \mathbf{if}\ \mathsf{e}\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2\ \mathbf{fi} \mid \mathbf{repeat}\ \mathsf{s}\ \mathbf{end} \mid \\
 & & [\iota]_{\mathsf{RM}} \mid [\iota]\ :=_{\mathsf{WM}}\ \mathsf{e} \mid \mathbf{cas}_{\mathsf{SM},\mathsf{FM}}(\iota, \mathsf{e}_1, \mathsf{e}_2) \\
\mathsf{s}_{\mathsf{RT}} & ::= & \mathbf{stuck} \mid \mathbf{par}\ \mathsf{s}_1\ \mathsf{s}_2 \\
 & & \\
\mathsf{RM} & ::= & \mathtt{sc} \mid \mathtt{acq} \mid \mathtt{con} \mid \mathtt{rlx} \mid \mathtt{na} \\
\mathsf{WM} & ::= & \mathtt{sc} \mid \mathtt{rel} \mid \mathtt{rlx} \mid \mathtt{na} \\
\mathsf{SM} & ::= & \mathtt{sc} \mid \mathtt{acqrel} \mid \mathtt{rel} \mid \mathtt{acq} \mid \mathtt{con} \mid \mathtt{rlx} \\
\mathsf{FM} & ::= & \mathtt{sc} \mid \mathtt{acq} \mid \mathtt{con} \mid \mathtt{rlx}
\end{array}
$$

Рис. 6. Синтаксис операций и выражений языка модели OpC11

так — в исполнении найдена гонка по данным или была попытка чтения из неинициализированной переменной. Другой конструкцией времени исполнения является $\mathbf{par}\ \mathsf{s}_1\ \mathsf{s}_2$, которая получается в результате редукции $\mathbf{spw}\ \mathsf{s}_1\ \mathsf{s}_2$ и представляет находящиеся в исполнении потоки. На шаге операционной семантики, который редуцирует $\mathbf{spw}\ \mathsf{s}_1\ \mathsf{s}_2$ в $\mathbf{par}\ \mathsf{s}_1\ \mathsf{s}_2$, происходит необходимая инициализация компонент машины OpC11.

Мета-переменная $\xi$ представляет динамическое состояние машины с точностью до программы. Вычисление программы $\mathsf{s}$ в машине OpC11 начинается со стартового состояния $\langle \mathsf{s}, \xi_{init} \rangle$, где $\xi_{init}$ содержит пустую память и пустой базовый фронт для единственного стартового потока. Переходы машины заданы в редукционном стиле [13], большинство из них имеет следующий вид:

$$\frac{\ldots}{\langle \mathsf{E}[\mathsf{s}], \xi \rangle \to \langle \mathsf{E}[\mathsf{s}'], \xi' \rangle}$$

Здесь E — это *редукционный контекст*, заданный так:

$$\mathsf{E} \;::=\; [\,] \mid \mathsf{x} := \mathsf{E};\ \mathsf{s} \mid \mathbf{par}\ \mathsf{E}\ \mathsf{s} \mid \mathbf{par}\ \mathsf{s}\ \mathsf{E}.$$

Если в момент исполнения программы запущено более одного потока, т.е. в программном выражении присутствует узел **par**, то программное выражение может быть недетерминированно разбито на контекст и подпрограмму.

Базовые правила семантики, которые не затрагивают операций над памятью, приведены на рисунке 7. Из них интерес представляют правила Spawn и Join, которые описывают старт и соединение дочерних потоков соответственно. Эти правила модифицируют компоненты состояния машины OpC11, которые являются локальными для потоков, например, базовый фронт и операционный буфер. Конкретные представления правил Spawn и Join зависят от многопоточных аспектов модели, которые определяют мета-функции spawn и join.

### 2.2.2 Представление памяти и фронтов

В базовом представлении состояние $\xi$ машины OpC11 включает в себя память $M$ и функцию $\psi^{\mathtt{rd}}$, которая по идентификатору потока $\pi$ возвращает его базовый фронт (см. рис. 8).

Память $M$ — это частичная функция, которая по локации и метке времени возвращает сохраненное значение и синхронизационный фронт. Разные аспекты модели используют фронты по-разному, но каждый фронт является частичной функцией, которая по локации возвращает некоторую метку времени. Функция базового фронта возвращает базовый фронт потока по его идентификатору $\pi$, где идентификатор потока — это список направлений налево (l) / направо (r), который показывает, как найти подвыражение потока во всем выражении программы, проходя по вершинам **par**. Это путь уникально идентифицирует поток. Для вычисления пути по редукционному контексту используется вспомогательная функция path.

Стартующие дочерние потоки наследуют базовый фронт родительского потока, что в терминах самой простой версии функции spawn определяется следующим образом:

$$\mathsf{spawn}(\mathsf{E}, \langle M, \psi^{\mathtt{rd}}\rangle) \triangleq \langle M, \psi^{\mathtt{rd}}[\pi\, l \mapsto \mathit{view}_{\mathtt{rd}}, \pi\, r \mapsto \mathit{view}_{\mathtt{rd}}]\rangle,$$

где $\pi = \mathsf{path}(\mathsf{E})$ и $\mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi)$. Когда потоки соединяются, базовый фронт их родительского потока становится покомпонентным максимумом базовых фронтов

Subst

$$\frac{}{\langle \mathsf{E}[\mathsf{x} := v;\ \mathsf{s}], \xi\rangle \to \langle \mathsf{E}[\mathsf{s}[\mathsf{x}/v]], \xi\rangle}$$

If-False

$$\frac{}{\langle \mathsf{E}[\mathtt{if}\ 0\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}], \xi\rangle \to \langle \mathsf{E}[\mathsf{s}_2]], \xi\rangle}$$

If-True

$$\frac{n \neq 0}{\langle \mathsf{E}[\mathtt{if}\ n\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}], \xi\rangle \to \langle \mathsf{E}[\mathsf{s}_1]], \xi\rangle}$$

Repeat-Unroll

$$\frac{\mathsf{x} - \text{новая переменная}}{\begin{array}{c}\langle \mathsf{E}[\mathtt{repeat}\ \mathsf{s}\ \mathtt{end}], \xi\rangle \to \\ \langle \mathsf{E}[\mathsf{x} := \mathsf{s};\ \mathtt{if}\ \mathsf{x}\ \mathtt{then}\ \mathsf{x}\ \mathtt{else}\ \mathtt{repeat}\ \mathsf{s}\ \mathtt{end}\ \mathtt{fi}], \xi\rangle\end{array}}$$

Spawn

$$\frac{\xi' = \mathsf{spawn}(\mathsf{E}, \xi)}{\langle \mathsf{E}[\mathtt{spw}\ \mathsf{s}_1\ \mathsf{s}_2], \xi\rangle \to \langle \mathsf{E}[\mathtt{par}\ \mathsf{s}_1\ \mathsf{s}_2], \xi'\rangle}$$

Join

$$\frac{\xi' = \mathsf{join}(\mathsf{E}, \xi)}{\langle \mathsf{E}[\mathtt{par}\ v_1\ v_2], \xi\rangle \to \langle \mathsf{E}[(v_1, v_2)], \xi'\rangle}$$

Choice-Fst

$$\frac{}{\langle \mathsf{E}[\mathsf{EU}[\mathtt{choice}\ \mathsf{e}_1\ \mathsf{e}_2]], \xi\rangle \to \langle \mathsf{E}[\mathsf{EU}[\mathsf{e}_1]], \xi\rangle}$$

Choice-Snd

$$\frac{}{\langle \mathsf{E}[\mathsf{EU}[\mathtt{choice}\ \mathsf{e}_1\ \mathsf{e}_2]], \xi\rangle \to \langle \mathsf{E}[\mathsf{EU}[\mathsf{e}_2]], \xi\rangle}$$

Рис. 7. Базовые правила OpC11 MM

| | | | |
|---|---|---|---|
| Состояние | $\xi$ | ::= | $\langle M, \psi^{\mathtt{rd}} \rangle$ |
| Память | $M$ | ::= | $(\ell, \tau) \rightharpoonup \langle v, \mathit{view} \rangle$ |
| Функция базового фронта | $\psi^{\mathtt{rd}}$ | ::= | $\pi \rightharpoonup \mathit{view}$ |
| Фронт | $\mathit{view}$ | ::= | $\ell \rightharpoonup \tau$ |
| Идентификатор потока | $\pi$ | — | $(l \mid r)^*$ |
| Метка времени | $\tau$ | $\in$ | $\mathbb{N}$ |

Рис. 8. Базовое состояние машины OpC11

Read-Uninit

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) = \bot}{\langle \mathsf{E}[[\ell]_{\mathsf{RM}}], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

CAS-Uninit

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) = \bot}{\langle \mathsf{E}[\mathbf{cas}_{\mathsf{SM},\mathsf{FM}}(\ell, \mathsf{e}_1, \mathsf{e}_2)], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

Рис. 9. Правила чтения из неинициализированной локации

дочерних потоков:

$$\mathsf{join}(\mathsf{E}, \langle \mathsf{s}, \psi^{\mathtt{rd}} \rangle) = \langle \mathsf{s}, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}^l_{\mathtt{rd}} \sqcup \mathit{view}^r_{\mathtt{rd}}] \rangle,$$

где $\pi = \mathsf{path}(\mathsf{E})$, $\mathit{view}^l_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi\, l)$ и $\mathit{view}^r_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi\, r)$.

Далее определяется первый набор правил о “плохих” сценариях поведения — тех, которые включают чтение из неинициализированной локации (см. рис. 9). Эти правила срабатывают в том случае, если поток, пытающийся прочитать значение из локации $\ell$, не осведомлен ни об одной записи, т.е. его базовый фронт не определён для $\ell$. В частном случае эти правила применимы, если в локацию не было сделано ни одной записи.

### 2.2.3 Высвобождающие и приобретающие обращения

Правила, описывающие поведение высвобождающих записей и приобретающих чтений, приведены на рис. 10. Высвобождающая запись добавляет в память новую запись $(\ell, \tau) \mapsto (v, \mathit{view})$, где $v$ — записываемое значение, а $\mathit{view}$ — фронт добавленного сообщения, который далее может быть использован для синхрони-

WRITE-RELEASE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = Next\tau(M, \ell) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view} = \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau] \quad \xi' = \langle M[(\ell, \tau) \mapsto (v, \mathit{view})], \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}] \rangle}{\langle \mathsf{E}[[\ell] \ :=_{\mathtt{rel}} \ v], \xi \rangle \to \langle \mathsf{E}[v], \xi' \rangle}$$

READ-ACQUIRE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad M(\ell, \tau) = (v, \mathit{view}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) \leqslant \tau \quad \xi' = \langle M, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}_{\mathtt{rd}} \sqcup \mathit{view}] \rangle}{\langle \mathsf{E}[[\ell]_{\mathtt{acq}}], \xi \rangle \to \langle \mathsf{E}[v], \xi' \rangle}$$

Рис. 10. Правила высвобождающей записи и приобретающего чтения

зации с другими потоками. Сам фронт сообщения является обновлённым на $[\ell@\tau]$ базовым фронтом соответствующего потока.

Правило обработки приобретающего чтения является недетерминированным, поскольку выбирает из памяти запись $(\ell, \tau) \mapsto (v, \mathit{view})$, метка времени $\tau$ которой не меньше значения базового фронта потока по этой локации $\mathit{view}_{\mathtt{rd}}(\ell)$. Значение $v$ заменяет инструкцию чтения внутри редукционного контекста $\mathsf{E}$, а базовый фронт потока $\mathit{view}_{\mathtt{rd}}$ комбинируется с прочитанным фронтом сообщения *view*. Здесь и далее правила для CAS-инструкций[4] опущены; они могут быть найдены в реализации интерпретатора [98].

### 2.2.4 `sc`-инструкции

В состояние машины OpC11 для представления правил обработки `sc`-инструкций нужно добавить дополнительную компоненту — глобальный `sc`-*фронт* $\mathit{view}^{\mathtt{sc}}$:

$$\xi \ ::= \ \langle \ldots, \mathit{view}^{\mathtt{sc}} \rangle$$

Этот фронт по локации определяет максимальную метку времени сообщения в эту локацию, которое было сделано с помощью `sc`-записи.

[4]CAS-инструкция (compare-and-set) является частным случаем атомарного чтения и записи (RMW). Эта инструкция прочитывает значение из локации, сравнивает его с ожидаемым значением и в случае совпадения записывает новое значение в локацию.

WriteSC

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{sc}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = \mathit{Next}\tau(M,\ell) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view} = \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau] \quad \xi' = \langle M[(\ell,\tau) \mapsto (v, \mathit{view})], \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}], \mathit{view}^{\mathtt{sc}}[\ell \mapsto \tau]\rangle}{\langle \mathsf{E}[[\ell] \ :=_{\mathtt{sc}} \ v], \xi\rangle \to \langle \mathsf{E}[v], \xi'\rangle}$$

ReadSC

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{sc}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad M(\ell,\tau) = (v, \mathit{view}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathsf{max}(\mathit{view}_{\mathtt{rd}}(\ell), \mathit{view}^{\mathtt{sc}}(\ell)) \leqslant \tau \quad \xi' = \langle M, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}_{\mathtt{rd}} \sqcup \mathit{view}], \mathit{view}^{\mathtt{sc}}\rangle}{\langle \mathsf{E}[[\ell]_{\mathtt{acq}}], \xi\rangle \to \langle \mathsf{E}[v], \xi'\rangle}$$

Рис. 11. Правила обработки `sc`-инструкций

Правила обработки `sc`-инструкций приведены на рис. 11. Они совпадают с правилами для высвобождающих записей и приобретающих чтений с рис. 10 с точностью до фрагментов, выделенных серым цветом. Так, правило `sc`-записи обновляет `sc`-фронт машины, чтобы по целевой локации $\ell$ он возвращал метку времени добавленной записи, а правило `sc`-чтения проверяет, что метка времени выбранного сообщения не меньше, чем значение `sc`-фронта по целевой локации.

### 2.2.5 Неатомарные обращения

Для обработки неатомарных чтений и записей в состояние машины добавляется глобальный `na`-фронт:

$$\xi \ ::= \ \langle \ldots, \mathit{view}^{\mathtt{na}}\rangle.$$

Аналогично `sc`-фронту, `na`-фронт по локации возвращает максимальную метку времени `na`-записи в неё, а сам фронт обновляется в правиле `na`-записи.

`na`-фронт предназначен для поиска гонок по данным, которые включают неатомарные обращения к памяти. Так, когда поток выполняет `na`-обращения (см. правила ReadNA и WriteNA на рис. 12), происходит проверка того, что поток осведомлён о последнем сообщении в целевую локацию. Если данное требование не выполняется, то в программе присутствует гонка, и программа имеет неопределённое поведение (см. правила ReadNA-stuck1 и WriteNA-stuck1). Кроме того, если поток выполняет необязательно неатомарное обращение к локации $\ell$, о по-

WriteNA

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = \mathit{Next}\tau(M, \ell) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view} = \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau] \quad \mathit{view}_{\mathtt{rd}}(\ell) \equiv \mathsf{LastTS}(M, \ell) \quad \xi' = \langle M[(\ell, \tau) \mapsto (v, ())], \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}], \mathit{view}^{\mathtt{na}}[\ell \mapsto \tau] \rangle}{\langle \mathsf{E}[[\ell] \ :=_{\mathtt{na}} \ v], \xi \rangle \to \langle \mathsf{E}[v], \xi' \rangle}$$

ReadNA

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = \mathsf{LastTS}(M, \ell) \quad M(\ell, \tau) = (v, \mathit{view}) \quad \tau \equiv \mathit{view}_{\mathtt{rd}}(\ell) \quad \xi' = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle}{\langle \mathsf{E}[[\ell]_{\mathtt{na}}], \xi \rangle \to \langle \mathsf{E}[v], \xi' \rangle}$$

ReadNA-stuck1

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) \neq \mathsf{LastTS}(M, \ell)}{\langle \mathsf{E}[[\ell]_{\mathtt{na}}], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

WriteNA-stuck1

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) \neq \mathsf{LastTS}(M, \ell)}{\langle \mathsf{E}[[\ell] \ :=_{\mathtt{na}} \ v], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

ReadNA-stuck2

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) < \mathit{view}^{\mathtt{na}}(\ell)}{\langle \mathsf{E}[[\ell]_{\mathsf{RM}}], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

WriteNA-stuck2

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \mathit{view}^{\mathtt{na}} \rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) < \mathit{view}^{\mathtt{na}}(\ell)}{\langle \mathsf{E}[[\ell] \ :=_{\mathsf{WM}} \ v], \xi \rangle \to \langle \mathbf{stuck}, \xi_{\mathit{init}} \rangle}$$

Рис. 12. Правила обработки неатомарных обращений и идентификации гонок по данным

$$\textsc{Read-Consume}\quad \frac{\xi = \langle M, \psi^{\mathtt{rd}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \qquad M(\ell, \tau) = (v, \mathit{view}) \qquad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \qquad \mathit{view}_{\mathtt{rd}}(\ell) \leqslant \tau \qquad \xi' = \langle M, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau]]\rangle}{\langle \mathsf{E}[[\ell]_{\mathtt{con}}], \xi\rangle \to \mathsf{annotate}(v, \mathit{view}, \pi, \langle \mathsf{E}[v], \xi'\rangle)}$$

Рис. 13. Правило обработки потребляющего чтения

следней `na`-записи в которую он не осведомлён, т.е. его базовый фронт $\mathit{view}_{\mathtt{rd}}$ по $\ell$ меньше, чем $\mathit{view}^{\mathtt{na}}(\ell)$, то это также является состоянием гонки (см. правила ReadNA-stuck2 и WriteNA-stuck2).

В отличие от рассмотренных ранее высвобождающих и `sc`-записей, `na`-запись не может быть использована для синхронизации (даже как часть высвобождающей цепочки), поэтому её сообщение имеет пустой фронт. Аналогично, `na`-чтение игнорирует фронт прочитанного им сообщения.

### 2.2.6 Потребляющие чтения

В отличие от приобретающего чтения, которое обновляет базовый фронт потока, и этим влияет на все последующие экземпляры чтения в потоке, потребляющее чтение влияют только на экземпляры, целевой адрес которых зависит от него. Аналогичным образом отличается влияние приобретающего и потребляющего чтений на последующие CAS-инструкции.

Для поддержки потребляющих чтений добавлена возможность во время исполнения программы аннотировать инструкции чтения и CAS дополнительным фронтом *view*, полученным из потребляющего чтения:

$$\begin{aligned} \mathsf{s}_{\mathsf{RT}} &::= \ldots \mid [\iota]_{\mathsf{RM},\,\mathit{view}} \mid \mathbf{cas}_{\mathsf{SM},\mathsf{FM},\,\mathit{view}}(\iota, v_1, v_2) \\ \beta &::= \ldots \mid \mathit{read}\langle \mathsf{x}, \iota, \mathsf{RM}, \mathit{view}\rangle \end{aligned}$$

При исполнении аннотированной инструкции базовый фронт потока временно увеличивается на фронт *view*.

При исполнении инструкции потребляющего чтения происходит аннотирование всех последующих зависимых инструкций с помощью соответствующего фронта сообщения (см. рис. 13).

Read-Relaxed

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \psi^{\mathtt{wr}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad M(\ell,\tau) = (v, \mathit{view}) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view}_{\mathtt{rd}}(\ell) \leqslant \tau \quad \xi' = \langle M, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}_{\mathtt{rd}} \sqcup [\ell@\tau]], \psi^{\mathtt{wr}}\rangle}{\langle \mathsf{E}[[\ell]_{\mathtt{rlx}}], \xi\rangle \to \langle \mathsf{E}[v], \xi'\rangle}$$

Write-Relaxed

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \psi^{\mathtt{wr}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = \mathit{Next}\tau(M,\ell) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view} = \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau] \quad \tau_{\mathtt{rel}} = \psi^{\mathtt{wr}}(\pi)(\ell) \quad (\_, \mathit{view}_{\mathtt{sync}}) = M(\ell, \tau_{\mathtt{rel}}) \quad \xi' = \langle M[(\ell,\tau) \mapsto (v, \mathit{view}_{\mathtt{sync}}[\ell \mapsto \tau])], \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}], \psi^{\mathtt{wr}}\rangle}{\langle \mathsf{E}[[\ell] :=_{\mathtt{rlx}} v], \xi\rangle \to \langle \mathsf{E}[v], \xi'\rangle}$$

Write-Release'

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \psi^{\mathtt{wr}}\rangle \quad \pi = \mathsf{path}(\mathsf{E}) \quad \tau = \mathit{Next}\tau(M,\ell) \quad \mathit{view}_{\mathtt{rd}} = \psi^{\mathtt{rd}}(\pi) \quad \mathit{view} = \mathit{view}_{\mathtt{rd}}[\ell \mapsto \tau] \quad \mathit{view}'_{\mathtt{wr}} = \psi^{\mathtt{wr}}(\pi)[\ell \mapsto \tau] \quad \xi' = \langle M[(\ell,\tau) \mapsto (v, \mathit{view})], \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}], \psi^{\mathtt{wr}}[\pi \mapsto \mathit{view}'_{\mathtt{wr}}]\rangle}{\langle \mathsf{E}[[\ell] :=_{\mathtt{rel}} v], \xi\rangle \to \langle \mathsf{E}[v], \xi'\rangle}$$

Рис. 14. Правила расслабленных обращений

### 2.2.7 Расслабленные обращения

Исполнение расслабленных инструкций чтения не обновляет базовый фронт потока на фронт прочитанного сообщения (см. правило Readext−Relaxed на рис. 14), и это отличает расслабленные инструкции от приобретающих.

Для поддержки расслабленных записей в состояние машины OpC11 нужно внести дополнительную компоненту $\psi^{\mathtt{wr}}$:

$$\xi \;::=\; \langle \ldots, \psi^{\mathtt{wr}}\rangle,$$

где $\psi^{\mathtt{wr}}$ — это функция, которая по идентификатору потока возвращает его *фронт записи* $\mathit{view}_{\mathtt{wr}}$. Фронт записи для каждой локации хранит метку времени последней высвобождающей записи, сделанной потоком в эту локацию. Когда поток выполняет расслабленную запись в локацию $\ell$ с меткой времени $\tau$ (см. правило Write-Relaxed на рис. 14), то в добавляемое сообщение записывается фронт, равный комбинации $[\ell@\tau]$ и фронта сообщения, на который указывает $\mathit{view}_{\mathtt{wr}}(\ell)$. Для

поддержки описанного механизма нужно изменить правило обработки высвобождающей записи (см. правило Write-Release′ на рис. 14): после выполнения данного правила фронт записи потока по целевой локации возвращает метку времени добавленного сообщения. Аналогично надо изменить другие подобные правила, в частности, WriteSC.

Кроме того, мета-функции, связанные с запуском и соединением потоков, также требуют нового определения:

$$\begin{aligned}\mathsf{spawn}(\mathsf{E}, \langle M,\psi^{\mathtt{rd}},\psi^{\mathtt{wr}}\rangle) &= \langle M, \psi^{\mathtt{rd}}[\pi\, l \mapsto \mathit{view}_{\mathtt{rd}}, \pi\, r \mapsto \mathit{view}_{\mathtt{rd}}],\\ &\qquad \psi^{\mathtt{wr}}[\pi\, l \mapsto \bot, \pi\, r \mapsto \bot]\rangle\\ \mathsf{join}(\mathsf{E}, \langle M, \psi^{\mathtt{rd}}, \psi^{\mathtt{wr}}\rangle) &= \langle M, \psi^{\mathtt{rd}}[\pi \mapsto \mathit{view}^{l}_{\mathtt{rd}} \sqcup \mathit{view}^{r}_{\mathtt{rd}}], \psi^{\mathtt{wr}}[\pi \mapsto \bot]\rangle\end{aligned}$$

Дочерние потоки при запуске, также как и родительский после соединения, не наследуют фронты записи, т.к. согласно C/C++11 MM [4] высвобождающие цепочки не могут выходить за пределы потока напрямую.

### 2.2.8 Отложенные операции и спекулятивное исполнение

Поддержка отложенных операций и спекулятивного исполнения в машине OpC11 осуществляется с помощью двух следующих компонент состояния:

$$\xi \;::=\; \langle \ldots, \varphi, \gamma\rangle.$$

Функция $\varphi$ по идентификатору потока $\pi$ возвращает его иерархический операционный буфер $\alpha$ отложенный операций $\beta$:

$$\begin{aligned}\varphi &::= \pi \rightharpoonup \alpha\\ \alpha &::= \beta^*\\ \beta &::= \mathit{read}\langle \mathsf{x},\iota,\mathsf{RM}\rangle \mid \mathit{write}\langle \mathsf{x},\iota,\mathsf{WM},\mathsf{e}\rangle \mid \mathit{bind}\langle \mathsf{x},\mathsf{e}\rangle \mid \mathit{if}\langle \mathsf{x},\mathsf{e},\alpha,\alpha\rangle\end{aligned}$$

Каждая отложенная операция $\beta$ обладает уникальным идентификатором — символьным значением $\mathsf{x}$. Отложенные операции чтения $\mathit{read}\langle \mathsf{x},\iota,\mathsf{RM}\rangle$ хранят выражение для вычисления целевой локации $\iota$[5], а также модификатор чтения $\mathsf{RM}$. В случае отложенной записи также сохраняется выражение $\mathsf{e}$, результат вычисления которого будет записан как целевое значение. Для представления отложенного присваивания в локальную переменную используется конструкция $\mathit{bind}\langle \mathsf{x}, \mathsf{e}\rangle$,

[5]Целевая локация отложенной операции может зависеть от других, ещё не вычисленных, отложенных операций.

которая может быть использована для того, чтобы не форсировать вычисление отложенного чтения:

$$a \ :=_{\texttt{rlx}} \ [x]; b \ := \ a + 1; \ldots$$

Для представления информации о спекулятивном исполнении веток условного оператора используется конструкция $if\langle \mathsf{x}, \mathsf{e}, \alpha_1, \alpha_2 \rangle$, где $\mathsf{e}$ является представлением условия, а $\alpha_1$ и $\alpha_2$ — вложенных буферов отложенных операций веток **`then`** и **`else`** соответственно. Редукционный контекст $\mathsf{E}\alpha$, внутри которого возможно спекулятивное исполнение инструкций, представлен следующей грамматикой:

$$\begin{aligned} \mathsf{E}\alpha \ ::= \ & [\,] \mid \mathsf{x} := \mathsf{E}\alpha;\ \mathsf{s} \mid \texttt{if}\ \mathsf{x}\ \texttt{then}\ \mathsf{E}\alpha\ \texttt{else}\ \mathsf{s}\ \texttt{fi} \mid \\ & \texttt{if}\ \mathsf{x}\ \texttt{then}\ \mathsf{s}\ \texttt{else}\ \mathsf{E}\alpha\ \texttt{fi} \end{aligned}$$

У конкретного подвыражения редукционного контекста символическое значение $\mathsf{x}$, представляющее условие спекулятивно исполняемой конструкции, должно совпадать с идентификатором соответствующей записи $if\langle \mathsf{x}, \mathsf{e}, \alpha_1, \alpha_2 \rangle$ в буфере отложенных операций.

Вторая добавленная компонента состояния, $\gamma$, представляет ограничения, накладываемые на приобретающие чтения, и является списком троек вида $\langle \ell, \tau, \mathsf{x} \rangle$. Присутствие тройки $\langle \ell, \tau, \mathsf{x} \rangle$ в списке $\gamma$ запрещает чтение из памяти сообщения $(\ell, \tau)$, пока отложенная операция с символическим значением $\mathsf{x}$ не будет выполнена.

В ходе исполнения программы любая инструкция чтения, записи или присваивания может быть отложена, даже если она находится в условном операторе — для этого в операционный буфер добавляется соответствующий элемент. Правила откладывания исполнения инструкций приведены на рис. 15.

После того, как операция была отложена, она может быть выполнена — см. правила на рис. 16. В этих правилах для записи и чтения опущены стандартные части, связанные с обновлением базового фронта и фронта записи. Рассмотрим подробнее правило Write-Resolve. С его помощью (недетерминированно) выбирается элемент из буфера отложенных операций, соответствующий некоторой инструкции записи, причём этот элемент обязан содержаться на первом уровне буфера. Без последнего требования семантика допускала бы “значения из воздуха”. Кроме того, перед выбранным элементом списка не должно быть элементов, которые находятся с ним в конфликте, таких как приобретающее чтение или запись в ту же локацию. Из новой компоненты отложенных операций убирается соответствующий элемент ($\varphi' = \mathsf{removeAndUpdate}(\varphi, \pi, write\langle \mathsf{x}, \ell, \mathsf{WM}, v \rangle)$),

WRITE-POSTPONE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \varphi, \gamma\rangle \qquad \mathsf{x} \text{ — новое символическое значение} \qquad \alpha = \varphi(\pi) \qquad \xi' = \langle M, \psi^{\mathtt{rd}}, \varphi[\pi \mapsto \mathsf{append}(\alpha, \mathsf{E}\alpha, \mathit{write}\langle \mathsf{x}, \iota, \mathsf{WM}, \mathsf{e}\rangle)], \gamma\rangle}{\langle \mathsf{E}[\mathsf{E}\alpha[[\iota] :=_{\mathsf{WM}} \mathsf{e}]], \xi\rangle \to \langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{x}]], \xi'\rangle}$$

READ-POSTPONE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \varphi, \gamma\rangle \qquad \mathsf{x} \text{ — новое символическое значение} \qquad \alpha = \varphi(\pi) \qquad \xi' = \langle M, \psi^{\mathtt{rd}}, \varphi[\pi \mapsto \mathsf{append}(\alpha, \mathsf{E}\alpha, \mathit{read}\langle \mathsf{x}, \iota, \mathsf{RM}\rangle)], \gamma\rangle}{\langle \mathsf{E}[\mathsf{E}\alpha[[\iota]_{\mathsf{RM}}]], \xi\rangle \to \langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{x}]], \xi'\rangle}$$

LET-POSTPONE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \varphi, \gamma\rangle \qquad \mathsf{e} \text{ зависит от отложенной операции} \qquad \mathsf{x} \text{ — новое символическое значение} \qquad \alpha = \varphi(\pi) \qquad \xi' = \langle M, \psi^{\mathtt{rd}}, \varphi[\pi \mapsto \mathsf{append}(\alpha, \mathsf{E}\alpha, \mathit{let}\langle \mathsf{x}, \mathsf{e}\rangle)], \gamma\rangle}{\langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{x}' := \mathsf{e};\ \mathsf{s}]], \xi\rangle \to \langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{x}'/\mathsf{x}]], \xi'\rangle}$$

Рис. 15. Правила откладывания исполнения инструкций чтения, записи и присваивания

обновляются фронты зависимых сообщений согласно $\gamma$-компоненте ($M' = \mathsf{updateSync}(\mathsf{x}, \mathsf{WM}, \mathit{view}, \gamma, M)$), а из самой $\gamma$-компоненты удаляются зависимости ($\gamma' = \mathsf{updateDep}(\mathsf{x}, \mathsf{WM}, \psi^{\mathtt{wr}}, \ell, \tau, \gamma, \varphi(\pi))$).

Отдельно стоит отметить правило, которое позволяет вынести отложенные инструкции записи, повторяющиеся в ветках спекулятивно исполняемого условного оператора, на предыдущий уровень вложенности буфера отложенных операций (см. рис. 17).

Правила обработки условного оператора (начало и завершение спекулятивного исполнения) приведены на рис. 18.

## 2.3 Интерпретация и тестирование модели

Интерпретатор для OpC11 MM [98] был разработан на языке программирования Racket [16, 17] с использованием средства разработки интерпретаторов PLT/Redex [18,19]. Средство PLT/Redex позволяет описать интерпретатор в стиле, использованном нами для описания модели — как множества правил переписывания состояния некоторой абстрактной машины. Также это средство использует

WRITE-RESOLVE

$$\frac{\xi = \langle M, \dots, \varphi, \gamma\rangle \qquad \dots \qquad write\langle \mathsf{x}, \ell, \mathsf{WM}, v\rangle \text{ находится на первом уровне } \varphi(\pi) \text{ и не конфликтует с предшествующими элементами списка} \qquad \varphi' = \mathsf{removeAndUpdate}(\varphi, \pi, write\langle \mathsf{x}, \ell, \mathsf{WM}, v\rangle) \qquad \gamma' = \mathsf{updateDep}(\mathsf{x}, \mathsf{WM}, \psi^{\mathtt{wr}}, \ell, \tau, \gamma, \varphi(\pi)) \qquad M' = \mathsf{updateSync}(\mathsf{x}, \mathsf{WM}, \mathit{view}, \gamma, M) \qquad \xi' = \langle M'[(\ell, \tau) \mapsto (v, \mathit{view})], \dots, \varphi'[v/\mathsf{x}], \gamma'\rangle}{\langle \mathsf{s}, \xi\rangle \to \langle \mathsf{s}[v/\mathsf{x}], \xi'\rangle}$$

READ-RESOLVE

$$\frac{\xi = \langle M, \dots, \varphi, \gamma\rangle \qquad \dots \qquad read\langle \mathsf{x}, \ell, \mathsf{RM}\rangle \text{ находится в } \varphi(\pi) \text{ и не конфликтует с предшествующими элементами} \qquad \varphi' = \mathsf{removeAndUpdate}(\varphi, \pi, read\langle \mathsf{x}, \ell, \mathsf{RM}\rangle) \qquad \gamma' = \gamma \setminus \{\langle \_, \_, \mathsf{x}\rangle\} \qquad \xi' = \langle M, \dots, \varphi'[v/\mathsf{x}], \gamma'\rangle}{\langle \mathsf{s}, \xi\rangle \to \langle \mathsf{s}[v/\mathsf{x}], \xi'\rangle}$$

LET-RESOLVE

$$\frac{\xi = \langle M, \psi^{\mathtt{rd}}, \varphi, \gamma\rangle \qquad bind\langle \mathsf{x}, v\rangle \text{ находится в } \varphi(\pi) \qquad \varphi' = \mathsf{removeAndUpdate}(\varphi, \pi, bind\langle \mathsf{x}, v\rangle) \qquad \xi' = \langle M, \psi^{\mathtt{rd}}, \varphi'[v/\mathsf{x}], \gamma\rangle}{\langle \mathsf{s}, \xi\rangle \to \langle \mathsf{s}[v/\mathsf{x}], \xi'\rangle}$$

Рис. 16. Правила по выполнению отложенных инструкций чтения, записи и присваивания

идеи редукционной семантики [13, 18, 19] и имеет встроенные механизмы разбиения термов на редукционный контекст и подтерм.

Реализация базовых правил целевого языка интерпретации, описанных в разделах 2.2.1 и 2.2.2, а также вспомогательных функций, занимает 2070 строк кода, реализация различных аспектов C/C++11 MM (разделы 2.2.3–2.2.8) — 1310 строк. Код тестов, описанных в разделах 2.3.1 и 2.3.2, занимает 3130 строк.

### 2.3.1 “Лакмусовые” тесты

Проверка адекватности OpC11 MM и её сравнение с C/C++11 MM были проведены на наборе т.н. *“лакмусовых” тестов* (litmus tests) — небольших программ,

WRITE-PROMOTE

$$\frac{\begin{array}{c}\xi = \langle M, \psi^{\mathrm{rd}}, \varphi, \gamma\rangle \qquad if\langle \mathsf{x}'', \mathsf{e}, \alpha_1, \alpha_2\rangle \text{ внутри } \varphi(\pi) \\ write\langle \mathsf{x}, \ell, \mathsf{WM}, v\rangle \text{ в списке } \alpha_1 \\ \text{и не конфликтует с предшествующими элементами списка} \\ write\langle \mathsf{x}', \ell, \mathsf{WM}, v\rangle \text{ в списке } \alpha_2 \\ \text{и не конфликтует с предшествующими элементами списка} \\ \varphi' = \mathsf{promote}(\mathsf{x}, \mathsf{x}', \varphi) \qquad \gamma' = \gamma[\mathsf{x}'/\mathsf{x}]\end{array}}{\langle \mathsf{s}, \xi\rangle \to \langle \mathsf{s}[\mathsf{x}'/\mathsf{x}], \xi'\rangle}$$

Рис. 17. Правило по переносу отложенной записи на предыдущий уровень вложенности буфера отложенных операций

IF-SPECULATION-INIT

$$\frac{\begin{array}{c}\xi = \langle M, \psi^{\mathrm{rd}}, \varphi, \gamma\rangle \\ \mathsf{e} \text{ зависит от отложенной операции} \qquad \mathsf{x} \text{ — новое символическое значение} \\ \alpha = \varphi(\pi) \qquad \xi' = \langle M, \psi^{\mathrm{rd}}, \varphi[\pi \mapsto \mathsf{append}(\alpha, \mathsf{E}\alpha, if\langle \mathsf{x},\mathsf{e}, \langle\rangle, \langle\rangle\rangle)], \gamma\rangle\end{array}}{\begin{array}{c}\langle \mathsf{E}[\mathsf{E}\alpha[\mathtt{if}\ \mathsf{e}\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}]], \xi\rangle \to \\ \langle \mathsf{E}[\mathsf{E}\alpha[\mathtt{if}\ \mathsf{x}\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}]], \xi'\rangle\end{array}}$$

IF-RESOLVE-TRUE

$$\frac{\begin{array}{c}\xi = \langle M, \psi^{\mathrm{rd}}, \varphi, \gamma\rangle \qquad if\langle \mathsf{x}, z, \alpha_1, \alpha_2\rangle \in \varphi(\pi) \\ z \neq 0 \qquad \varphi' = \varphi[if\langle \mathsf{x}, z, \alpha_1, \alpha_2\rangle/\alpha_1] \qquad \xi' = \langle M, \psi^{\mathrm{rd}}, \varphi', \gamma\rangle\end{array}}{\langle \mathsf{E}[\mathsf{E}\alpha[\mathtt{if}\ \mathsf{x}\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}]], \xi\rangle \to \langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{s}_1]], \xi'\rangle}$$

IF-RESOLVE-FALSE

$$\frac{\begin{array}{c}\xi = \langle M, \psi^{\mathrm{rd}}, \varphi, \gamma\rangle \\ if\langle \mathsf{x}, 0, \alpha_1, \alpha_2\rangle \in \varphi(\pi) \qquad \varphi' = \varphi[if\langle \mathsf{x}, z, \alpha_1, \alpha_2\rangle/\alpha_2] \qquad \xi' = \langle M, \psi^{\mathrm{rd}}, \varphi', \gamma\rangle\end{array}}{\langle \mathsf{E}[\mathsf{E}\alpha[\mathtt{if}\ \mathsf{x}\ \mathtt{then}\ \mathsf{s}_1\ \mathtt{else}\ \mathsf{s}_2\ \mathtt{fi}]], \xi\rangle \to \langle \mathsf{E}[\mathsf{E}\alpha[\mathsf{s}_2]], \xi'\rangle}$$

Рис. 18. Правила начала и завершения спекулятивного исполнения условного оператора

которые являются показательными примерами различных видов взаимодействия потоков. Эти тесты активно используются в литературе [4, 66, 99–101] для обсуждения свойств моделей памяти. При тестировании нам нужно было проверить, что ожидаемые результаты исполнения программы и её исходы совпадают. Для этого по каждому тесту средствами PLT/Redex строился полный, т.е. включающий в себя все возможные сценарии поведения теста, граф состояний и переходов в соответствии с правилам, задающим интерпретатор OpC11. Далее из этого графа извлекалось множество финальных состояний, которое сравнивалось с ожидаемыми результатами.

В табл. 1 предоставлена информация о “лакмусовых” тестах в контексте OpC11 MM. Столбцы VF–JN показывают, какие аспекты модели требуются для поддержки нужного поведения тестов. В таблице использованы следующие сокращения: VF (viewfront) — базовый фронт, $\psi^{\mathtt{rd}}$; WF (write-fronts) — фронт записи, $\psi^{\mathtt{wr}}$; SCF (sc-front) — sc-фронт, $view^{\mathtt{sc}}$; NAF (na-front) — na-фронт, $view^{\mathtt{na}}$; PO (postponed operations) — отложенные операции, $\alpha$; ARR (acquire read restrictions) — ограничение приобретающих чтений, $\gamma$; CR (consume-reads) — маркировка, связанная с потребляющими чтениями; JN (joining threads) — соединение потоков с не пустыми буферами. Последний столбец C11 представляет информацию о том, полностью ли совпадают результаты OpC11 MM и C/C++11 MM.

В таблице тесты разделены на группы; в каждой группе тесты имеют схожую структуру, но различаются модификаторами доступа. Код тестов и информация об ожидаемом поведении тестов размещён в приложении A.

## Расхождения с MM C/C++11 MM

Для того, чтобы на тестах LB-acq-rlx и LB-acq-rlx-join поддержать те же результаты исполнения, что и в C/C++11 MM, нужно позволить инструкции расслабленной записи, которая следует за инструкцией приобретающего чтения, быть исполненной перед этим чтением. Как и все слабые сценарии поведения, такой сценарий может быть следствием либо компиляторной оптимизации, либо оптимизирующего исполнения программы на процессоре. В общем случае перестановка приобретающего чтения с какой бы то ни было последующей инструкцией небезопасна, т.к. приобретающее чтение может быть частью синхронизации, следовательно в результате перестановки может быть изменено отношение

“предшествует” hb. Поэтому промышленные компиляторы, такие как GCC [102] и Clang [103], эту оптимизацию не реализуют. В то же время все корректные схемы компиляции инструкций приобретающего чтения для основных процессорных архитектур (x86, Power, ARM) также гарантируют, что сценарий поведения, который разрешается в C/C++11 MM, но запрещён в OpC11 MM, не наблюдается на этих архитектурах.

В нашей модели не позволяется выполнять не по порядку инструкции, относящиеся к одной локации. Как следствие, наша модель не разрешает слабый сценарий поведение теста ARM-weak (см. приложение A.10), в результате которого $a = 1$. При этом такой результат возможен в C/C++11 MM. Подробное описание данного теста приведено в главе 3.

Кроме того, наша модель не совпадает с C/C++11 MM на примерах со “значениями из воздуха” (OOTA-lb, OOTA-if), что является преимуществом.

### 2.3.2 Проверка модели на примере RCU-структуры

Кроме “лакмусовых” тестов OpC11 MM была также проверена в контексте тестирования и отладки многопоточной структуры данных RCU (read-copy-update) [104, 105]. RCU-структура является одной из стандартных стратегий реализации неблокирующего доступа к связной структуре данных, такой как список или дерево, для одного писателя и множества читателей. Автор диссертационного исследования реализовал RCU-структуру для односвязного списка на основе *освобождения памяти в момент затишья* (quiescent state-based reclamation, QSBR) [106]. Центральной идеей RCU-структуры является то, каким образом писатель обновляет её узлы. Если писатель собирается изменить некоторый узел в списке, то вместо того, чтобы изменить его непосредственно, он создаёт копию старого узла, обновляет её, а потом изменяет ссылки в структуре таким образом, чтобы они указывали на новый узел вместо старого. Далее писатель ждёт, пока все читатели не перестанут использовать старый узел, после чего память, отведенная под узел, может быть переиспользована или освобождена. Для корректности алгоритма необходима правильная синхронизация между писателем и читателями: писатель обновляет ссылки на узлы с помощью высвобождающей записи, тогда как читатели обходят список с помощью приобретающих чтений. Это гарантирует, что читатели не увидят частичные изменения в структуре данных.

| **Название теста** | VF | WF | SCF | NAF | PO | ARR | CR | JN | **C11** |
|---|---|---|---|---|---|---|---|---|---|
| SB-rel-acq | ✓ | | | | | | | | ✓ |
| SB-sc | ✓ | | ✓ | | | | | | ✓ |
| SB-sc-rel | ✓ | | ✓ | | | | | | ✓ |
| SB-sc-acq | ✓ | | ✓ | | | | | | ✓ |
| LB-rlx | ✓ | | | | ✓ | | | | ✓ |
| LB-rel-rlx | ✓ | | | | ✓ | | | | ✓ |
| LB-acq-rlx | ✓ | | | | ✓ | | | | ✗ |
| LB-rel-acq-rlx | ✓ | | | | ✓ | ✓ | | | ✓ |
| LB-rlx-use | ✓ | | | | ✓ | | | | ✓ |
| LB-rlx-let | ✓ | | | | ✓ | | | | ✓ |
| LB-rlx-join | ✓ | | | | ✓ | | | ✓ | ✓ |
| LB-rel-rlx-join | ✓ | | | | ✓ | | | ✓ | ✓ |
| LB-acq-rlx-join | ✓ | | | | ✓ | | | ✓ | ✗ |
| MP-rlx-na | ✓ | | | ✓ | | | | | ✓ |
| MP-rel-rlx-na | ✓ | | | ✓ | | | | | ✓ |
| MP-rlx-acq-na | ✓ | | | ✓ | | | | | ✓ |
| MP-rel-acq-na | ✓ | | | ✓ | | ✓ | | | ✓ |
| MP-rel-acq-na-rlx(_2) | ✓ | ✓ | | ✓ | | ✓ | | | ✓ |
| MP-con-na(_2) | ✓ | | | ✓ | | | ✓ | | ✓ |
| MP-cas-rel-acq-na | ✓ | | | ✓ | | ✓ | | | ✓ |
| MP-cas-rel-rlx-na | ✓ | | | ✓ | | | | | ✓ |
| CoRR-rlx | ✓ | | | | | | | | ✓ |
| CoRR-rel-acq | ✓ | | | | | | | | ✓ |
| IRIW-rlx | ✓ | | | | | | | | ✓ |
| IRIW-rel-acq | ✓ | | | | | | | | ✓ |
| IRIW-sc | ✓ | | ✓ | | | | | | ✓ |
| WRC-rlx | ✓ | | | | | | | | ✓ |
| WRC-rel-acq | ✓ | | | | | | | | ✓ |
| WRC-cas-rel | ✓ | | | | | ✓ | | | ✓ |
| WRC-cas-rlx | ✓ | | | | | | | | ✓ |
| OOTA-lb | ✓ | | | | ✓ | | | | ✗ |
| OOTA-if | ✓ | | | | ✓ | | | | ✗ |
| WR-rlx | ✓ | | | | ✓ | | | | ✓ |
| WR-rlx-rel | ✓ | | | | ✓ | ✓ | | | ✓ |
| WR-rel | ✓ | | | | ✓ | ✓ | | | ✓ |
| SE-simple | ✓ | | | | ✓ | | | | ✓ |
| SE-prop | ✓ | | | | ✓ | | | | ✓ |
| SE-nested | ✓ | | | | ✓ | | | | ✓ |
| ARM-weak | ✓ | | | | ✓ | | | | ✗ |
| Блокировка Деккера | ✓ | | | ✓ | | | | | ✓ |
| Блокировка Коэна [66] | ✓ | | | ✓ | | | | | ✓ |

Табл. 1. Результаты запуска интерпретатора ОрС11 ММ на “лакмусовых” тестах

**Программа:**

$[cw] :=_{\texttt{na}} 0; [cr1] :=_{\texttt{na}} 0; [cr2] :=_{\texttt{na}} 0; [lhead] :=_{\texttt{na}} \textbf{null};$

| | | |
|---|---|---|
| | $[sum11] :=_{\texttt{na}} 0;$ | $[sum21] :=_{\texttt{na}} 0;$ |
| | rcuOnline$(cw, cr1)$; | rcuOnline$(cw, cr2)$; |
| | traverse$(lhead, cur1, sum11)$; | traverse$(lhead, cur2, sum21)$; |
| $[a] :=_{\texttt{rlx}} (1, \textbf{null})$; | rcuOffline$(cw, cr1)$; | rcuOffline$(cw, cr2)$; |
| $[ltail] :=_{\texttt{na}} a$; | | |
| $[lhead] :=_{\texttt{rel}} a$; | $[sum12] :=_{\texttt{na}} 0$; | $[sum22] :=_{\texttt{na}} 0$; |
| append$(b, 10, ltail)$; | rcuOnline$(cw, cr1)$; | rcuOnline$(cw, cr2)$; |
| append$(c, 100, ltail)$; | traverse$(lhead, cur1, sum12)$; | traverse$(lhead, cur2, sum22)$; |
| updateSecondNode$(d, 1000)$ | rcuOffline$(cw, cr1)$; | rcuOffline$(cw, cr2)$; |
| | | |
| | $r11 :=_{na} [sum11]$; | $r21 :=_{na} [sum21]$; |
| | $r12 :=_{na} [sum12]$ | $r22 :=_{na} [sum22]$ |

**Функции:**

append$(loc, value, ltail) \triangleq$
- $[loc] :=_{\texttt{rlx}} (value, \textbf{null})$;
- $rt :=_{\texttt{na}} [ltail]$;
- $rtc :=_{\texttt{rlx}} [rt]$;
- $[rt] :=_{\texttt{rel}} (\textbf{fst}\, rtc, loc)$;
- $[ltail] :=_{\texttt{na}} loc$

updateSecondNode$(loc, value) \triangleq$
- $r1 :=_{\texttt{rlx}} [lhead]$;
- $r1c :=_{\texttt{rlx}} [r1]$;
- $r2 := \textbf{snd}\, r1c$;
- $r2c :=_{\texttt{rlx}} [r2]$;
- $r3 := \textbf{snd}\, r2c$;
- $[loc] :=_{\texttt{rel}} (value, r3)$;
- $[r1] :=_{\texttt{rel}} (\textbf{fst}\, r1c, loc)$;
- sync$(cw, cr1, cr2)$;
- $\textbf{delete}\, r2$

sync$(cw, cr1, cr2) \triangleq$
- $rcw :=_{\texttt{rlx}} [cw]$;
- $rcwn := rcw + 2$;
- $[cw] :=_{\texttt{rel}} rcwn$;
- syncWithReader$(rcwn, cr1)$;
- syncWithReader$(rcwn, cr2)$

traverse$(lhead, curNodeLoc, resLoc) \triangleq$
- $rh :=_{\texttt{acq}} [lhead]$;
- $[curNodeLoc] :=_{\texttt{na}} rh$;
- **repeat**
  - $rCurNode :=_{\texttt{na}} [curNodeLoc]$;
  - **if** $rCurNode \neq \textbf{null}$
  - **then** $rNode :=_{\texttt{acq}} [rCurNode]$;
    $rRes :=_{\texttt{na}} [resLoc]$;
    $rVal := \textbf{fst}\, rNode$;
    $[resLoc] :=_{\texttt{na}} rVal + rRes$;
    $[curNodeLoc] :=_{\texttt{na}} \textbf{snd}\, rNode$;
    0
  - **else** 1
  - **fi**
- **end**

syncWithReader$(rcwn, cr) \triangleq$
- $\textbf{repeat}\, r0 :=_{\texttt{acq}} [cr]; r0 >= rcwn\, \textbf{end}$

rcuOnline$(cw, cr) \triangleq$
- $r0 :=_{\texttt{acq}} [cw]$;
- $[cr] :=_{\texttt{rlx}} r0 + 1$

rcuOffline$(cw, cr) \triangleq$
- $r0 :=_{\texttt{rlx}} [cw]$;
- $[cr] :=_{\texttt{rel}} r0$

Рис. 19. Реализация алгоритма QSBR RCU

QSBR-реализация RCU-списка и использующая его клиентская программа приведена на рис. 19. В первой строке происходит инициализация счётчиков потоков: $cw$ – счётчик (первого) потока-писателя, $cr1$ и $cr2$ – счётчики второго и третьего потоков программы, которые являются читателями структуры. В той же строке происходит инициализация указателя $lhead$ на начало списка. Далее происходит запуск трёх потоков.

Поток-писатель добавляет в список три значения: 1, 10 и 100. Заметим, что поскольку в нашей модели не описывается выделение памяти, то для записи значений используются константные локации $a$, $b$ и $c$. После того, как поток записывает три значения, он заменяет второй элемент списка, в котором хранится значение 10, на новый узел $d$, в который записывается значение $1000$. При обновлении списка с помощью функции updateSecondNode поток-читатель вызывает функцию синхронизации sync, внутри которой происходит обновление счётчика $cw$, и ожидает обновления других потоков (вызовы функции syncWithReader).

Рассмотрим подробнее функцию append, которая используется для добавления второго и третьего значений в список. В этой функции сначала создаётся новый узел ($[loc] :=_{\texttt{rlx}} (value, \textbf{null})$), далее указатель на последний элемент списка записывается в переменную $rt$ ($rt :=_{\texttt{na}} [ltail]$), происходит разыменование указателя ($rtc :=_{\texttt{rlx}} [rt]$), после чего по указателю $rt$ записывается старое значение последнего элемента списка (**fst** $rtc$) и указатель на новый узел ($loc$). В конце обновляется указатель на последний элемент списка ($[ltail] :=_{\texttt{na}} loc$). Отметим, что в функции append все обращения к памяти, кроме $[rt] :=_{\texttt{rel}} (\textbf{fst}\ rtc, loc)$, являются неатомарными или расслабленными. Это связано с тем, что локация $ltail$ и переменные $rt$ и $rtc$ локальны для первого потока. При этом высвобождающая запись $[rt] :=_{\texttt{rel}} (\textbf{fst}\ rtc, loc)$ делает так, что потоки-читатели, обходящие список с помощью приобретающих чтений в функции traverse, становятся осведомлёнными о записи в новый узел ($[loc] :=_{\texttt{rlx}} (value, \textbf{null})$) в тот момент, когда получают указатель на ячейку $loc$.

Потоки-читатели по два раза обходят список, создаваемый писателем, и вычисляют сумму его элементов (переменные $r11, r12, r21$ и $r22$). Перед началом обхода списка они вызывают функцию rcuOnline, которая выполняет две задачи. Во-первых, эта функция выполняет приобретающее чтение из $cw$, которое синхронизирует поток с некоторой версией списка. Во-вторых, с помощью увеличения счётчика $cr$ поток сигнализирует писателю о том, что он начал обход списка.

Симметрично, читатель обновляет свой счётчик и уведомляет поток-писатель после обхода списка с помощью функции rcuOffline.

## Дополнительная инфраструктура для тестирования RCU-структуры

С помощью запускаемой операционной семантики можно провести динамический анализ приведённой выше программы, а именно, построить пространство состояний исполнения программы в OpC11 MM и проверить его свойства, такие как отсутствие гонок по данным. Тем не менее, такой подход не является реалистичным, поскольку пространство состояний для этой программы очень велико. Это связано с тем, что модель имеет три аспекта недетерминированности:

– планирование потоков;
– отложенные операции;
– недетерминированное чтение даже из фиксированной локации.

Все эти аспекты приводят к комбинаторному взрыву пространства состояний.

Для того, чтобы сделать динамический анализ возможным на практике (хотя бы в ограниченном варианте), диссертантом была реализована версия модели, которая строит некоторый случайный путь в пространстве состояний исполнения программы. На первом шаге к текущему состоянию машины недетерминированно применяются все возможные правила. Далее проверяется, есть ли в получившемся множестве **`stuck`**-состояние, которое сигнализирует о том, что программа имеет неопределённое поведение. Если такого состояние нет, то из множества случайным образом выбирается новое состояние, и процедура повторяется. Имея такое представление модели, мы можем проводить свойство-ориентированное тестирование программы [107].

Дополнительно, в язык задания модели был добавлен оператор **`delete`**, который “удаляет” переданную ему локацию. Данный оператор может быть использован для проверки того, что некоторая локация, начиная с некоторого момента исполнения программы, больше не используется, что является одним из свойств RCU-структуры. Для задания семантики этому оператору в состояние машины OpC11 был добавлен дополнительный список “удалённых” локаций. Обращение к локации из этого списка приводит к **`stuck`**-состоянию.

| # | $r11$ | $r12$ | $r21$ | $r22$ | **stuck** | **время** (сек.) |
|---|---|---|---|---|---|---|
| 1 | 0 | 111 | 111 | 1101 | ✓ | 25.2 |
| 2 | 0 | 1 | 111 | 1101 | | 21.4 |
| 3 | 0 | 0 | 0 | 0 | | 12.9 |
| 4 | 0 | 1101 | 11 | 11 | | 25.4 |
| 5 | 0 | 1101 | 0 | 0 | | 16.3 |
| 6 | 0 | 11 | 0 | 0 | ✓ | 17.5 |
| 7 | 0 | 0 | 0 | 1101 | | 22.1 |
| 8 | 1 | 1 | 0 | 0 | | 16.5 |
| 9 | 0 | 1101 | 1 | 1101 | | 19.2 |
| 10 | 0 | 1101 | 111 | 1101 | | 26.4 |
| 11 | 1 | 1101 | 1 | 1 | | 23.8 |
| 12 | 0 | 0 | 111 | 1101 | ✓ | 20.5 |
| 13 | 11 | 1101 | 0 | 0 | | 21.5 |
| 14 | 0 | 111 | 0 | 111 | ✓ | 21.5 |
| 15 | 0 | 0 | 11 | 1101 | | 22.1 |
| 16 | 0 | 0 | 0 | 0 | | 16.0 |
| 17 | 0 | 0 | 0 | 1101 | | 18.1 |
| 18 | 1 | 1 | 0 | 0 | | 22.2 |
| 19 | 1 | 1101 | 1 | 1 | | 26.0 |
| 20 | 1 | 1101 | 0 | 0 | | 20.1 |

Табл. 2. Результаты тестирования модифицированной программы с RCU

## Тестирование и отладка RCU-структуры

С помощью рандомизированной модели автор диссертационного исследования протестировал приведенную RCU-структуру. На более чем 20 запусках программы не было получено **stuck**-состояния.

Далее, в программу была внесена ошибка синхронизации, а именно удалены вызовы функции syncWithReader, помеченные серым фоном на рис. 19. Кроме того, автором учитывались ещё два дополнительных критерия корректности программы. Во-первых, все значения $r11, r12, r21$ и $r22$ должны быть из множества $\{0, 1, 11, 111, 1101\}$, что гарантирует корректность прочтения списка. Во-вторых, должны выполняться неравенства $r11 \leqslant r12$ и $r21 \leqslant r22$, поскольку после каждого добавления и изменения в списке сумма его элементов увеличивается.

Тест был запущен 20 раз на компьютере с процессором Core i7 2.5Ghz, 8Gb оперативной памяти и операционной системой Linux. Каждый из запусков завершился менее чем за 27 секунд, и критерий корректности переменных $r*$ не нарушался (см. табл. 2). Тем не менее из-за того, что в модифицированной версии программы вызовы функции syncWithReader не предшествуют освобождению локации с помощью оператора **`delete`**, потоки-читатели обращаются к "удалённым" локациям, о чём свидетельствует графа **`stuck`**.

Аналогично, рандомизированное тестирование указывает на наличие ошибки, если в функции traverse приобретающее чтение внутри цикла **`repeat`** поменять на расслабленное. Это приводит к тому, что между читателем и писателем не происходит синхронизации, и позволяет читателю провести чтение из локаций $a$, $b$, $c$ и $d$, о которых он не осведомлён. Последнее приводит к **`stuck`**-состоянию по правилу Read-Uninit.

## 2.4 Выводы

В данной главе была представлена операционная модель памяти для C/C++11, OpC11 MM. Эта модель задана как множество аспектов, которые описывают различные особенности C/C++11. Ключевыми понятиями модели являются *фронты* и *операционные буфера*. Фронты используются для представления знания (осведомленности) потоков о текущем состоянии общей памяти, операционные буфера позволяют откладывать исполнение инструкций и производить спекулятивные вычисления.

OpC11 MM рассмотрена на наборе тестовых программ из литературы, а также на RCU-структуре, в результате был сделан следующий вывод: поведение модели совпадает с C/++11 MM на большинстве программ. OpC11 MM предлагает синтаксический способ обработки спекулятивного исполнения, что позволяет оставить модель вычисляемой, т.е. даёт возможность реализовать её интерпретатор. Но следует отметить, что по этой же причине модель не поддерживает некоторые компиляторные оптимизации, связанные с семантическими свойствами программы.

# Глава 3. Корректность компиляции из обещающей модели в операционную модель ARMv8 POP

В этой главе доказывается корректность эффективной схемы компиляции из подмножества обещающей модели памяти [11] в модель памяти ARMv8 POP [9]. Рассмотренное подмножество обещающей модели состоит из расслабленных обращений памяти, высвобождающего и приобретающего барьеров памяти.

## 3.1 Мотивация доказательства корректности компиляции

Обещающая модель (Promise MM) является операционной моделью памяти для языков программирования. Она похожа на OpC11 MM, описанную в предыдущей главе: память представляется как множество событий, упорядоченных метками времени, а ключевым механизмом модели также являются фронты. Существенным отличием обещающей модели от OpC11 MM является то, что первая предлагает семантический способ для эмуляции переупорядочивания инструкций: вместо того, чтобы откладывать операции, поток может *пообещать*, что он в будущем сделает определенную запись; обещанное сообщение становится доступным для чтения другими потоками. Как следствие, обещающая модель поддерживает больше оптимизаций и ближе к C/C++11 MM. Негативным отличием обещающей модели от OpC11 MM является то, что упомянутый механизм обещаний требует, чтобы после каждого шага исполнения обещающей модели происходила *сертификация* — механизм, гарантирующий отсутствие “значений из воздуха”. Сертификация является алгоритмически неразрешимой в случае, когда обещающая модель задана для языка, полного по Тьюрингу, и следовательно, в этом случае для обещающей невозможно разработать интерпретатор. Несмотря на это, научное сообщество рассматривает обещающую модель как кандидата на замену моделей памяти языков C/C++ и Java. Для того, чтобы модель действительно смогла стать частью стандартов этих языков, должна быть доказана корректность компиляции во все основные целевые платформы: x86, Power, ARM.

Модель памяти ARMv8 POP является операционной и описывает многопоточное поведение процессорной архитектуры ARMv8.0. В этой версии архитектуры появилось несколько новых инструкций, которые предназначены не только упростить компиляцию из языков C/C++11, но и повысить производительность целевых программ. Так, в частности, в модели появились отдельные инструкции

приобретающего чтения и высвобождающей записи. Кроме того, модель ARMv8 POP является очень слабой, т.е. позволяет слабые сценарии поведения, которые не наблюдаются в рамках других моделей памяти процессорных архитектур.

Для обещающей модели памяти ранее была доказана корректность компиляции в модели x86-TSO [7] и Power [6] в [11]. Доказательство было основано на том, что модели x86-TSO и Power представимы в виде набора локальных переупорядочиваний, для которых доказана корректность в обещающей модели памяти, над более строгими моделями. Для более строгих моделей верно, что всех их сценарии поведения могут быть воспроизведены в подмножестве обещающей модели, в котором не используется механизмы обещаний и, как следствие, сертификации.

Такой подход не применим для модели ARMv8 POP. Так, следующая программа в модели ARMv8 POP имеет слабый сценарий поведения с результатом $a = 1$:

$$\begin{array}{ccc|c|ccc} & & & [x] := 0; & [y] := 0; & & \\ a := [x]; & //1 & & b := [x]; & c := [y]; & & \\ [x] := 1 & & & [y] := b & [x] := c & & \end{array} \qquad \text{(ARM-weak)}$$

В этой программе любое переупорядочивание инструкций невозможно, следовательно упомянутый сценарий поведения воспроизводим в обещающей модели только с помощью механизма обещаний.

## 3.2 Описание моделей на примерах

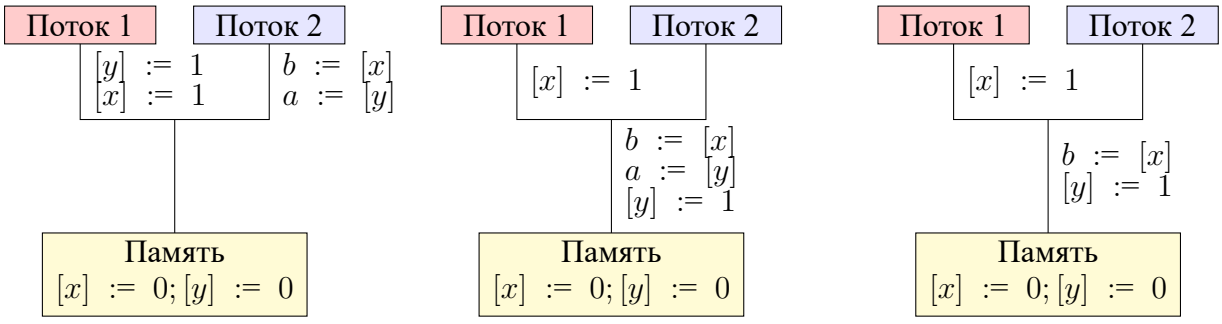
Мы начнём обсуждение ARMv8 POP и обещающей моделей с того, что покажем, как в них воспроизводится слабое поведение программы MP.

$$\begin{array}{ccc|c|ccc} & & & [x] := 0; & [y] := 0; & & \\ [x] := 1; & & & a := [y]; & //1 & & \\ [y] := 1 & & & b := [x] & //0 & & \end{array} \qquad \text{(MP)}$$

Схема компиляции из обещающей модели в ARMv8 POP, рассмотренная в доказательстве корректности компиляции, имеет следующий вид:

| | | | | |
|---|---|---|---|---|
| **Promise:** | $[x] :=_{\texttt{rlx}} a$ | $a :=_{\texttt{rlx}} [x]$ | $\mathsf{fence}(\texttt{acq})$ | $\mathsf{fence}(\texttt{rel})$ |
| **ARMv8 POP:** | $[x] := a$ | $a := [x]$ | $\mathsf{fence}(\texttt{ld})$ | $\mathsf{fence}(\texttt{sy})$ |

Она является биекцией на инструкциях обоих моделей, поэтому все программы в этой главе будут представлены в одном синтаксисе — синтаксисе модели ARMv8 POP.




а) б) в)

Рис. 20. Состояние подсистемы памяти ARMv8 POP при исполнении программы MP

### 3.2.1 Модель ARMv8 POP

Модель ARMv8 POP задана операционно, и, соответственно, существует некоторая абстрактная машина (*ARM-машина*) которая описывает сценарии поведения в рамках этой модели. ARM-машина состоит из двух компонент: *подсистемы управления* (thread subsystem) и *подсистемы памяти* (storage subsystem). До некоторой степени эти подсистемы можно рассматривать как набор вычислительных ядер (подсистема управления), которые, выполняя программу, посылают запросы в оперативную и кэш-память (подсистема памяти). Подсистема памяти в ARM-машине представлена как иерархическая система *буферов*, где каждый буфер является изменяемым списком запросов, а у основания иерархической системы находится *память* — функция из локаций в значения. Запросы бывают трёх видов: на чтение, на запись и барьер памяти.

Результат $[a = 1, b = 0]$ в ARM-машине может быть получен двумя способами. Во-первых, подсистема управления имеет право исполнять инструкции не по порядку (out-of-order execution). Так, первый поток в программе MP может отправить запрос на запись $[y] := 1$ в подсистему памяти до запроса $[x] := 1$, а второй поток, соответственно, может увидеть эти запросы в отличном от программного порядке. Во-вторых, подсистема памяти также может переупорядочить запросы внутри себя. Так, сначала оба потока могут отправить по два запроса в подсистему памяти (см. рис. 20а). После этого подсистема переупорядочивает независимые запросы $[x] := 1$ и $[y] := 1$, отправляет $[y] := 1$ в общий для двух потоков буфер, как и запросы второго потока (см. рис. 20б). Поскольку в общем буфере запросы $[y] := 1$ и $a := [y]$ находятся рядом, то подсистема памяти может послать ответ $[a \leftarrow 1]$ на запрос $a := [y]$ в подсистему управления (см. рис. 20в).

После этого запрос $[x] := 1$ отправляется в общий буфер, а затем все оставшиеся запросы отправляются в память, что приводит к результату $[a = 1, b = 0]$.

В разделе 1.2.2 было сказано, что аналогичный сценарий поведения в случае архитектуры Power может быть запрещен с помощью барьеров памяти. Это верно и для модели ARMv8 POP. Так, следующая программа не имеет сценария поведения с результатом $[a = 1, b = 0]$:

$$\begin{array}{l|l} [x] := 0; [y] := 0; & \\ [x] := 1; & a := [y]; \ //1 \\ \mathsf{fence}(\mathtt{sy}); & \mathsf{fence}(\mathtt{ld}); \\ [y] := 1 & b := [x] \ //0 \end{array} \qquad \text{(MP-SY-LD)}$$

*Полный барьер* $\mathsf{fence}(\mathtt{sy})$ в первом потоке заставляет подсистему управления отправить все три запроса в исходном порядке; кроме того, запрос, соответствующий полному барьеру, не может быть переупорядочен с другими запросами внутри подсистемы памяти[1]. Так, барьер $\mathsf{fence}(\mathtt{sy})$ гарантирует, что запрос $[y] := 1$ попадает в общий буфер подсистемы памяти только после $[x] := 1$. Во втором потоке *барьер на чтение* $\mathsf{fence}(\mathtt{ld})$ запрещает подсистеме управления отправлять запрос на чтение $b := [x]$ до того момента, пока она не получит ответ на запрос $a := [y]$. Таким образом гарантируется, что если $a = 1$, то $b = 1$.

В приведенных выше сценариях было важно, что подсистема памяти может переупорядочивать лишь некоторые пары сообщений. Какие пары могут быть переупорядочены, а какие не могут, определяется отношением $e_{\text{old}} \hookrightarrow e_{\text{new}}$:

**Определение 4.** Запросы $e_{\text{old}}$ и $e_{\text{new}}$ могут быть *переупорядочены*, что обозначается $e_{\text{old}} \hookrightarrow e_{\text{new}}$, если среди них нет полного барьера, и они оперируют над разными локациями.

Отметим, что в доказательстве корректности компиляции рассматривается ослабленная версия модели ARMv8 POP [9], которая допускает большее число сценариев поведения, поскольку в ней подсистема управления не отсылает запросы, соответствующие барьеру $\mathsf{fence}(\mathtt{ld})$, в подсистему памяти[2]. Поскольку нам

[1] Вместо полного барьера в этой программе можно использовать *барьер на запись* $\mathsf{fence}(\mathtt{st})$, однако барьер на запись не соответствует ни одной конструкции в обещающей модели, так же, как и в C/C++11 MM.

[2] Почему рассмотренная модель допускает большее число поведений? Предположим, что модель посылает запросы $\mathsf{fence}(\mathtt{ld})$ в память, но они могут быть переупорядочены со всеми остальными. Такая мо-

удалось доказать корректность компиляции для ослабленной версии модели, то корректность компиляции сохраняется и для исходной модели.

Теперь рассмотрим сценарий поведения программы ARM-weak, в котором получается результат $[a = 1, b = 1, c = 1]$:

$$\begin{array}{lll} & [x] := 0; [y] := 0; & \\ a := [x]; \ //1 & \| \ b := [x]; & \| \ c := [y]; \\ [x] := 1 & \| \ [y] := b & \| \ [x] := c \end{array} \qquad \text{(ARM-weak)}$$

Такой результат получается в модели ARMv8 POP, если у первого и второго потоков есть общий буфер, который не виден третьему потоку. Для получения результата $[a = 1, b = 1, c = 1]$ первый поток отправляет оба запроса (на чтение $a := [x]$ и на запись $[x] := 1$) в подсистему памяти, а второй поток отправляет запрос на чтение $b := [x]$, и после этого все запросы попадают в общий для первого и второго потоков буфер (см. рис. 21а). Поскольку запросы на чтение $b := [x]$ и на запись $[x] := 1$ находятся в одном буфере, и ближе к основной памяти находится запрос на запись, то подсистема памяти может послать ответ $[b \leftarrow 1]$ второму потоку, а сам поток после этого — послать запрос $[y] := 1$, который через общий буфер и переупорядочивание с запросами о локации $x$ в нём попадает в основную память (см. рис. 21б). Теперь третий поток посылает запрос на чтение $c := [y]$ (см. рис. 21в), который разрешается в общей памяти с ответом $[c \leftarrow 1]$. После этого третий поток посылает запрос на запись $[x] := 1$ (см. рис. 21г), который далее используется для отправки ответа на запрос чтения $a := [x]$, приводя к результату $[a = 1, b = 1, c = 1]$.

### 3.2.2 Абстрактная подсистема памяти: POP

В [9] подсистема памяти модели ARM представлена в двух вариантах: Flowing и POP (partial order propagation). Несмотря на то, что мы называем модель из [9] моделью ARMv8 POP, до этого момента в примерах использовалась подсистема памяти Flowing, поскольку она более наглядна. Тем не менее, у Flowing есть два ключевых недостатка, которые решаются в POP — более абстрактном варианте подсистемы памяти. Во-первых, вместо того, чтобы представлять буфер как список запросов и рассматривать в нём перестановки, математически удобнее

дель, очевидно, допускает больше сценариев поведения, чем оригинальная, накладывающая ограничения на переупорядочивание с fence(ld). Также очевидно, что полученная модель допускает те же сценарии, что и модель, которая не посылает запросы fence(ld).

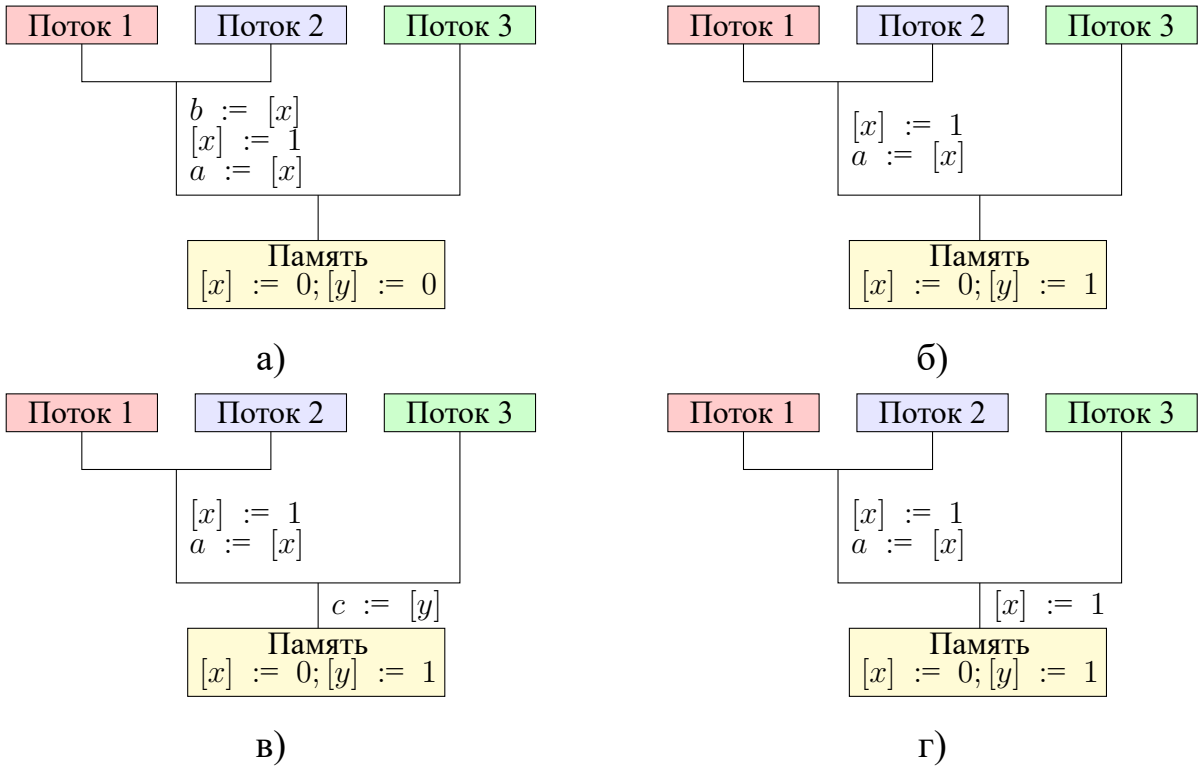



Рис. 21. Состояние подсистемы памяти ARMv8 POP при исполнении программы ARM-weak

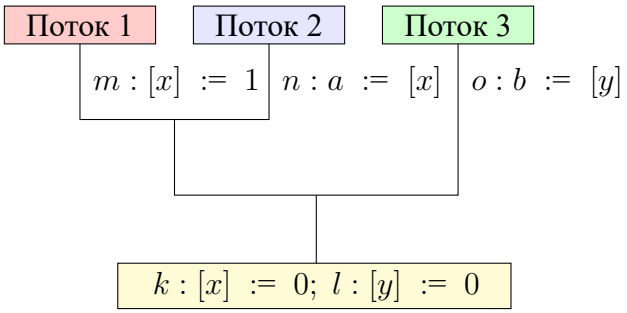
представлять его как частично упорядоченное множество. Во-вторых, подсистема Flowing требует введения конкретной топологии буферов, что при рассуждениях о всех сценариях поведения программы заставляет дополнительно рассматривать и варианты топологий.

В подсистеме POP нет линейных буферов и фиксированной топологии. Вместо этого состояние подсистемы представляется как тройка $\langle Evt, Ord, Prop \rangle$, где $Evt$ — это множество текущих запросов в подсистеме, $Ord$ — это частичный порядок на запросах из $Evt$, а $Prop$ — функция, которая по идентификатору потока возвращает множество “распространённых” на поток запросов. Если запросы $e$ и $e'$ упорядочены отношением $Ord$, т.е. $Ord(e, e')$, то мы пишем $e <_{Ord} e'$.

Для того, чтобы понять, как подсистема POP работает и соотносится с подсистемой Flowing, рассмотрим слабый сценарий поведения следующей программы с результатом $[a = 1, b = 1, c = 0]$:

$$[x] := 1; \left\| \begin{array}{ll} a := [x]; & // 1 \\ [y] := a & \end{array} \right\| \left\| \begin{array}{ll} b := [y]; & // 1 \\ c := [x + b * 0] & // 0 \end{array} \right. \qquad \text{(WRC-data-addr)}$$

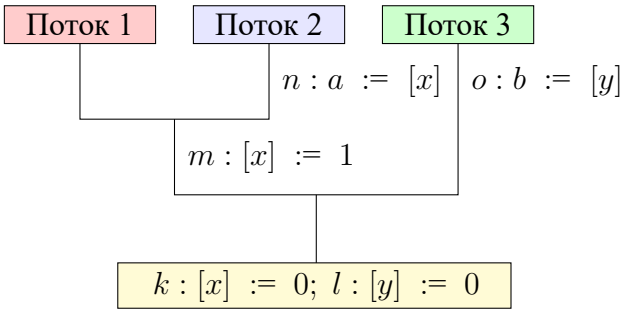
В данной программе подсистема управления модели ARMv8 POP во втором и третьем потоках не может отправить запросы, относящиеся ко второй строке, пока




а)

$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, n\}$$
$$Prop(T3) = \{k, l, o\}$$

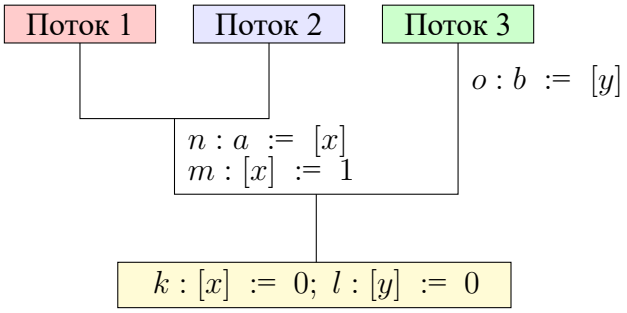
б)




в)

$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o), (m, n)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, n, m\}$$
$$Prop(T3) = \{k, l, o\}$$

г)




д)

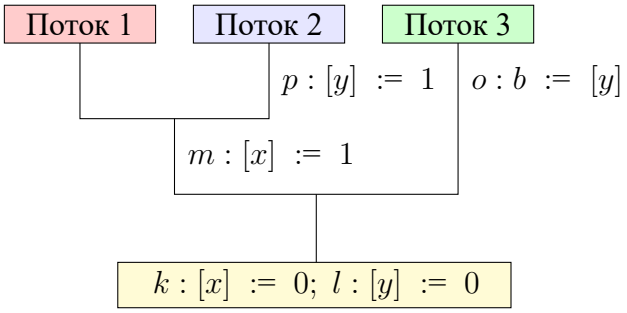
$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o), (m, n)\}$$
$$Prop(T1) = \{k, l, m, n\}$$
$$Prop(T2) = \{k, l, n, m\}$$
$$Prop(T3) = \{k, l, o\}$$

е)

Рис. 22. Состояние подсистем Flowing и POP при исполнении программы WRC-data-addr

не получит ответ на первые инструкции. Это связано с тем, что во втором потоке между инструкциями существует зависимость по данным, а в третьем — мнимая (fake) зависимость по адресу. Таким образом, эффект слабого сценария поведения в ARMv8 POP должен быть получен за счёт подсистемы памяти.

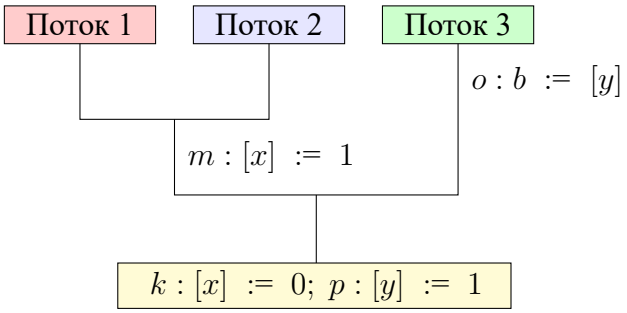
Для того, чтобы воспроизвести упомянутый результат, в рамках подсистемы Flowing нужно выбрать такую же топологию буферов, как и для слабого сценария поведения программы ARM-weak, т.е. должен присутствовать буфер, общий для первого и второго потоков.




ж)

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, m, p\}$$
$$Prop(T3) = \{k, l, o\}$$

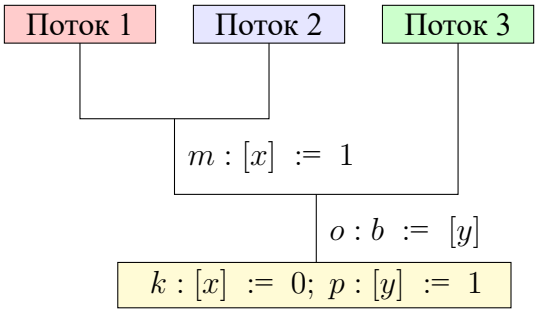
и)




к)

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p), (p, o)\}$$
$$Prop(T1) = \{k, l, m, p\}$$
$$Prop(T2) = \{k, l, m, p\}$$
$$Prop(T3) = \{k, l, o, p\}$$

л)




м)

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p), (p, o)\}$$
$$Prop(T1) = \{k, l, m, p, o\}$$
$$Prop(T2) = \{k, l, m, p, o\}$$
$$Prop(T3) = \{k, l, o, p\}$$

н)

Рис. 22. Состояния подсистем Flowing и POP при исполнении программы WRC-data-addr (продолжение)

Пусть все три потока отправили по одному запросу в подсистему памяти. Соответствующие состояния Flowing и POP представлены на рис. 22а и 22б. Заметим, что для краткости мы помечаем запросы в Flowing буквенными идентификаторами и используем их в описании состояния POP.

Когда поток $T$ отправляет запрос $e$ в подсистему памяти POP, во множество *Ord* добавляется по ребру $(e', e)$ для каждого запроса $e'$, о котором поток $T$ осведомлён (т.е. $e' \in \mathit{Prop}(T)$) и который также не может быть переупорядочен с $e$ (т.е. $e' \not\hookrightarrow e$). Поэтому на рис. 22б отношение *Ord* не пусто — в нём присутствуют ребра, связывающие отправленные потоками запросы с инициализирующими записями в локации.

Далее, подсистема памяти может перенести запрос $m : [x] := 1$ в буфер, общий для первого и второго потоков. В терминах подсистемы POP это означает, что второй поток оказывается осведомлён об этом запросе (см. рис. 22в и 22г). Запрос $m$ попадает во множество $\mathit{Prop}(T2)$, а в отношение *Ord* добавляется ребро $(m, n)$.

В общем случае, когда поток $T'$ оказывается осведомлён о запросе $e$ потока $T$, т.е., в терминах Flowing, $e$ попадает в видимый для $T'$ буфер, или, в терминах POP, $e$ становится элементом $\mathit{Prop}(T')$, то подсистема POP добавляет по ребру $(e, e')$ во множество *Ord* для каждого запроса $e' \in \mathit{Prop}(T') \backslash \mathit{Prop}(T)$, если запросы не могут быть переупорядочены (т.е. $e \not\hookrightarrow e'$) и между ними нет обратного ребра $(e', e)$ в отношении *Ord*. В данном сценарии программы WRC-data-addr запросы $m$ и $n$ не переупорядочиваемы, т.к. они оперируют одной и той же локацией $x$, поэтому на рис. 22г между ними появляется *Ord*-ребро.

Далее, запрос $n$ попадает в буфер, общий для первого и второго потоков (подсистема Flowing, рис. 22д), что соответствует тому, что первый поток становится осведомлённым о запросе $n$ (подсистема POP, рис. 22е). Теперь запрос $n : a := [x]$ непосредственно следует за запросом $m : [x] := 1$ в общем для первого и второго потоков буфере (подсистема Flowing), о запросах осведомлены одни и те же потоки, и между ними нет никакого запроса в отношении *Ord* (подсистема POP). Это означает, что и Flowing, и POP могут отправить ответ $[n \leftarrow 1]$ на запрос $n : a := [x]$. После получения ответа второй поток отправляет запрос $p : [y] := 1$ в подсистему памяти (см. рис. 22ж и 22и). Далее, подсистема памяти отправляет запрос $p : [y] := 1$ в общий буфер и в основную память (см. рис. 22к, подсистема Flowing), и после этого первый и третий потоки оказываются осведомлёнными о нём (см. рис. 22л, подсистема POP). Заметим, что поскольку в

подсистеме POP нет фиксированной топологии буферов, подсистема может осведомить первый поток о запросе $p$, а потом — третий поток, или наоборот.

Теперь запрос $o : b := [y]$ может попасть в общий для всех потоков буфер (см. рис. 22м и 22н) и затем в память, что приведёт к отправке ответа $[b \leftarrow 1]$ на него. После этого третий поток посылает запрос $c := [x]$. Этот запрос, попав в основную память, получает ответ $[c \leftarrow 0]$. В конце сценария поведения запрос $m : [x] := 1$ попадает в память. Итак, мы получили результат $[a = 1, b = 1, c = 0]$.

В конце исполнения программы в модели ARMv8 POP с подсистемой POP все выпущенные запросы записи и барьеров памяти попадают в основную память, и, соответственно, все потоки оказываются осведомлены о них. Следовательно, все записи в одну локацию тотально упорядочиваются отношением *Ord*. В рамках описанного ниже доказательства корректности компиляции этот порядок используется для введения меток времени на запросах записи в подсистеме памяти.

В доказательстве корректности компиляции используется именно подсистема POP, поскольку в [9] доказано, что подсистема POP допускает все сценарии поведения, возможные с подсистемой Flowing, и кроме того является более абстрактной.

### 3.2.3 Обещающая модель

Как уже было отмечено в этой главе, обещающая модель памяти, аналогично OpC11 MM, использует метки времени, базовые фронты и фронты сообщений. Так, слабый сценарий поведения программы MP, имеющий результат $[a = 1, b = 0]$, имеет абсолютно ту же структуру в обещающей модели, что и в OpC11 MM. Кратко проиллюстрируем его.

$$\begin{array}{ll} [x] := 0; & [y] := 0; \\ [x] := 1; & \left\| \begin{array}{l} a := [y]; \ //1 \\ b := [x] \ //0 \end{array} \right. \end{array} \qquad \text{(MP)}$$

После исполнения инициализирующих записей и старта потоков память и базовые фронты потоков имеют следующие значения:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После того, как левый поток выполнил обе записи, в памяти появляются два новых сообщения, и базовый фронт первого потока обновляется:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [x@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}].$$

Теперь второй поток может прочитать новое сообщение $\langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle$ и старое сообщение $\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle$, что приводит к результату $[a = 1, b = 0]$.

Рассмотрим, как слабый сценарий поведения с результатом $[a = 1, b = 0]$ программы MP-SY-LD запрещается в обещающей модели:

$$\begin{array}{l} [x] := 0; [y] := 0; \\ \begin{array}{l|l} [x] := 1; & a := [y]; \ //1 \\ \mathsf{fence}(\mathtt{sy}); & \mathsf{fence}(\mathtt{ld}); \\ [y] := 1 & b := [x] \ //0 \end{array} \end{array} \qquad \text{(MP-SY-LD)}$$

В рассматриваемой нами схеме компиляции барьеры $\mathsf{fence}(\mathtt{sy})$ и $\mathsf{fence}(\mathtt{ld})$ являются результатами компиляции высвобождающего и приобретающего барьеров соответственно. Для их поддержки в обещающей модели используются высвобождающий ($\mathit{view}_{\text{rel}}$) и приобретающий ($\mathit{view}_{\text{acq}}$) фронты потоков. Последний аналогичен приобретающему фронту в OpC11 MM.

После выполнения инициализирующих записей и старта потоков конфигурация абстрактной машины, реализующей обещающую модель (т.н. *обещающей машины*), выглядит следующим образом:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T1.\mathit{view}_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T1.\mathit{view}_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$$
$$T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После того, как первый поток выполнил запись $[x] := 1$, в память добавляется новое сообщение, а базовый и приобретающий фронты потока обновляются соответствующей меткой времени:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\};$$
$$T1.\mathit{view}_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.\mathit{view}_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.\mathit{view}_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$$
$$T2.\mathit{view}_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.\mathit{view}_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

При этом фронтом сообщения становится высвобождающий фронт $\mathit{view}_{\text{rel}}$ первого потока, увеличенный на метку времени сообщения по целевой локации.

После того, как первый фронт выполняет высвобождающий барьер, который в программе представлен его скомпилированным вариантом fence(sy), высвобождающий фронт $view_{\text{rel}}$ потока становится равен базовому фронту:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\};$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{0}];\quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{0}];\quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}];\quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}];\quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После этого первый поток выполняет вторую запись по тем же правилам, что и первую:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\};$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}];\quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}];\quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

Теперь второй поток может выполнить чтение из добавленного сообщения, получая $a = 1$. При этом базовый фронт потока $view_{\text{cur}}$ обновляется на $[y@\mathbf{1}]$, а приобретающий $view_{\text{acq}}$ — на фронт сообщения, $[x@\mathbf{1}, y@\mathbf{1}]$, как это было в OpC11 MM:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\};$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}];\quad T2.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

После выполнения приобретающего барьера, который в программе представлен его скомпилированным вариантом fence(ld), базовый фронт $view_{\text{cur}}$ второго потока становится равным приобретающему фронту $view_{\text{acq}}$:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\};$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]\quad T2.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}];\quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}].$$

Теперь второй поток не может прочитать старое сообщение $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$, т.к. $T2.view_{\text{cur}}(x) = \mathbf{1} > \mathbf{0}$. Таким образом, результат $[a = 1, b = 0]$ запрещается.

Перейдем к рассмотрению слабого сценария поведения программы ARM-weak с результатом $[a = 1, b = 1, c = 1]$ в обещающей модели[3].

$$\begin{array}{ccc|c|c} & & [x] := 0; & [y] := 0; & \\ a := [x]; & //1 & b := [x]; & c := [y]; & \\ [x] := 1 & & [y] := b & [x] := c & \end{array} \qquad \text{(ARM-weak)}$$

Для того, чтобы получить интересующий нас результат, обещающая модель, в соответствии со своим названием, использует *механизм обещаний*. Так, в любой момент исполнения поток может сделать одно из двух следующих действий: либо выполнить следующую инструкцию, либо *пообещать* сделать некоторую запись в будущем. Когда поток $T$ обещает сделать запись, он добавляет в память машины сообщение $\langle \ell : val@\tau, view \rangle$, где $\tau$ — уникальная метка времени локации $\ell$, которая больше, чем $T.view_{\text{cur}}(\tau)$. После этого сообщение становится доступным для чтения другими потоками по обычным правилам, но сам поток $T$ не может читать из этого сообщения, пока не выполнит обещание. Для того, чтобы проверять данное условие, у каждого потока имеется множество обещанных, но ещё не выполненных сообщений $T.promises$. После каждого перехода в обещающей машине поток, совершивший данный переход, должен пройти *сертификацию*, т.е. показать, что может выполнить все свои обещания в текущем состоянии памяти, будучи запущенным в изоляции. Процесс сертификации позволяет решить проблему “значений из воздуха”.

Результат $[a = 1, b = 1, c = 1]$ для программы ARM-weak в обещающей машине получается следующим образом. После выполнения инициализирующих записей и старта дочерних потоков первый поток обещает сообщение $\langle x : 1@\mathbf{2}, [x@\mathbf{2}, y@\mathbf{0}] \rangle$[4]. После этого второй поток прочитывает данное сообщение, получая $b = 1$, и выполняет запись, добавляя сообщение $\langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}] \rangle$ в память. Далее, третий поток прочитывает добавленное сообщение, получая $c = 1$, и добавляет в память сообщение $\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}] \rangle$. Первый поток прочитывает это сообщение, получая $a = 1$, и после этого выполняет данное ранее обещание $\langle x : 1@\mathbf{2}, [x@\mathbf{2}, y@\mathbf{0}] \rangle$, исполняя инструкцию записи $[x] := 1$.

[3]Стоит отметить, что такой результат для программы ARM-weak невозможен в OpC11 MM, но возможен в C/C++11 MM.

[4]Отметим ещё одно отличие обещающей модели и OpC11 MM. В OpC11 MM метки времени являются натуральными числами, и когда поток делает запись сообщения, метка времени сообщения равна увеличенной на единицу максимальной метке по соответствующей локации. В обещающей модели метки времени представлены положительными вещественными числами, и при добавлении нового сообщения его метка времени может быть произвольной с точностью до базового фронта соответствующего потока.

## 3.3 Основные идеи доказательства корректности компиляции

В разделе 1.2.2 было представлено следующее определение понятия корректности компиляции: все результаты, возможные для скомпилированной программы в целевой модели памяти, должны быть также возможны и для изначальной программы в исходной модели памяти. Поскольку рассматриваемая в этой главе схема компиляции является биекцией, мы можем считать, что изначальная и скомпилированная программы совпадают. Соответственно, нам надо показать, что для любой программы в синтаксисе модели ARMv8 POP и её произвольного сценария поведения в модели ARMv8 POP существует сценарий поведения этой программы с тем же результатом в обещающей модели.

До этого момента в рассмотренных примерах под результатом сценария мы понимали значение локальных переменных после исполнения программы. Тем не менее, для формального определения корректности компиляции между обещающей и ARMv8 POP моделей под результатом сценария удобнее понимать *финальное состояние памяти* — функцию, которая по локации возвращает значение её последней записи. Так, в случае модели ARMv8 POP под последней записью в локацию $\ell$ понимается запрос $[\ell] := v$, попавший последним в основную память, а в случае обещающей модели — сообщение $\langle \ell : v@\tau, \mathit{view} \rangle$, где $\tau$ является максимумом среди всех остальных сообщений в $\ell$.

Также отметим, что далее будут рассматриваться только конечные сценарии поведения в рамках модели ARMv8 POP, т.к. для рассмотрения не завершающихся сценариев пришлось бы ввести ограничения на спекулятивное исполнение программ, которого в модели нет. Так, следующая программа в модели ARMv8 POP имеет сценарий поведения, в котором соответствующий поток бесконечно посылает запросы на чтение и не получает на них ответы:

$$a := [x];$$
$$\textsf{if } a \neq 0 \textsf{ goto } -1$$

Здесь инструкция $\textsf{if } cond \textsf{ goto } shift$ имеет следующую семантику: если выражение $cond$ вычисляется к истинному (не нулевому) значению, то к указателю инструкций потока прибавляется значение $shift$, иначе — прибавляется единица, т.е. управление переходит к следующей инструкции.

Центральная теорема данной главы, которая утверждает корректность компиляции, формулируется следующим образом:

**Теорема 1.** Для любой программы *Prog* и её сценария поведения в модели ARMv8 POP, $\mathit{Prog} \vdash \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}$, где $\mathbf{s}$ — финальное состояние, $\mathsf{Final}^{\text{ARM}}(\mathbf{s}, \mathit{Prog})$, существует сценарий поведения этой программы в обещающей модели, $\mathit{Prog} \vdash \mathbf{p}^{\text{init}} \xrightarrow[\text{Promise}]{}{}^{*} \mathbf{p}$, где $\mathbf{p}$ — финальное состояние $\mathsf{Final}^{\text{Promise}}(\mathbf{p}, \mathit{Prog})$; при этом состояния памяти в $\mathbf{s}$ и $\mathbf{p}$ совпадают, $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

Стандартной техникой для доказательства таких утверждений является *симуляция* [14, 15, 108]. Для этого определяется *отношение симуляции*, связывающее состояния соответствующих абстрактных машин, и показывается, что если текущие состояния машин связаны этим отношением, и симулируемая машина (в нашем случае это ARM-машина) делает шаг, то симулирующая машина (в нашем случае это обещающая машина) может сделать ноль и более шагов так, чтобы новые состояния машин были также связаны отношением симуляции.

В доказательстве автор диссертации также использовал технику симуляции, однако не напрямую. Это связано со следующими особенностями нашего случая.

1. В каждый момент исполнения в обещающей машине имеется тотальный порядок на сообщениях в конкретную локацию — метки времени. В то же время в модели ARMv8 POP тотальный порядок на запросах записи в локацию может быть гарантированно определён только в конце исполнения.
2. В модели ARMv8 POP тот факт, что запрос чтения был удовлетворён из конкретного запроса записи, накладывает ограничения на дальнейшее исполнение, однако эти ограничения не определяются явно, как это делается с помощью фронтов в обещающей модели.
3. ARM-машина может исполнять разные типы инструкций не по порядку, в частности, инструкции записи и инструкции чтения, тогда как обещающая машина — только инструкции записи (механизм обещаний).

Отношение симуляции обычно предоставляет достаточно сильную связь между компонентами машин. Так, в нашем случае естественно было бы ожидать, что это отношение связывает каждое конкретное сообщение в памяти обещающей машины с некоторым запросом записи в подсистеме памяти ARM-машины. Тем не менее, первые две особенности, упомянутые выше, не позволяют этого сделать. Чтобы преодолеть данное ограничение, автор диссертационного исследования вводит промежуточную машину ARM+$\tau$. Она является версией ARM-машины, в которой каждый запрос записи $w$, попавший в подсистему памяти, до-

полнительно помечен следующей информацией: (i) меткой времени; (ii) множеством запросов записи и барьеров памяти $S$, о которых гарантированно становится осведомлён поток, прочитавший из запроса $w$; (iii) фронтом, представляющим множество $S$. При этом метка времени запроса $w$ отражает то, в каком порядке относительно других запросов записи в ту же локацию запрос $w$ попадёт в основную память машины. Тем не менее, в тот момент, когда запрос записи попадает в подсистему памяти, обычная ARM-машина не может гарантировать никакого конкретного места в этом порядке для запроса. Для решения этой проблемы в правилах переходов машины ARM+τ введены дополнительные ограничения, которые гарантируют ацикличность объединения частичного порядка *Ord* и порядка, полученного с помощью меток времени. Наличие данных ограничений означает, что машина ARM+τ потенциально имеет меньше сценариев поведения, чем изначальная машина. Следовательно, для того, чтобы использовать ARM+τ как промежуточную машину в доказательстве корректности компиляции, нам нужно доказать, что ARM+τ может симулировать ARMv8 POP, т.е. для каждой конкретной программы ARM+τ имеет не меньший набор сценариев поведения, чем ARMv8 POP (теорема 3, раздел 3.6.4).

Из-за третьей особенности мы не можем ввести непосредственное соотношение между шагами ARMv8 POP и обещающей машин. Так, рассмотрим следующую программу:

$$[x] := 1;$$
$$\mathsf{fence}(\mathtt{ld});$$
$$a := [y];$$
$$[z] := 1$$

ARM-машина может исполнить её следующим образом. Сначала она выполняет барьер памяти $\mathsf{fence}(\mathtt{ld})$ (шаг 1), этого отправляет запрос на чтение $a := [y]$, на который приходит ответ (шаг 2), посылает запрос на запись $[z] := 1$ (шаг 3), и после этого — запрос $[x] := 1$ (шаг 4). Обещающая машина не может исполнить эту программу в том же порядке, поэтому ей позволяется *запаздывать* (lag) относительно ARM-машины при симуляции. Так, пока ARM-машина делает первый и второй шаги, обещающая машина запаздывает и не делает ничего. На третий шаг ARM-машины обещающая машина отвечает обещанием записать сообщение $\langle z : 1@\_,\_\rangle$. На четвертый шаг обещающая машина отвечает серией шагов: (i) обещает записать сообщение $\langle x : 1@\_,\_\rangle$, (ii) выполняет это обещание, (iii) ис-

$$
\begin{array}{lll}
cmds & : List\ S & \\
S & ::= reg\ :=\ [expr] & \\
 & \mid [expr_0]\ :=\ expr_1 & \\
 & \mid \mathsf{fence}(fmod_{\mathrm{ARM}}) & \\
 & \mid \mathsf{if}\ expr\ \mathsf{goto}\ k & \\
 & \mid reg\ :=\ expr \mid \mathsf{nop} & \\
fmod_{\mathrm{ARM}} & ::= \mathtt{sy} \mid \mathtt{ld} & \\
expr & ::= reg \mid \ell \mid uop\ expr & \\
 & \mid bop\ expr_0\ expr_1 & \\
 & : \mathsf{e} & \\
reg : Reg & - a, b, c, \ldots & \text{(локальные переменные)} \\
\ell : Loc & - x, y, z, \ldots & \text{(локации)} \\
uop, bop & - \text{арифметические операции} & \\
k & \in \mathbb{Z} &
\end{array}
$$

Рис. 23. Синтаксис программ в модели ARMv8 POP

полняет приобретающий барьер (поскольку $\mathsf{fence}(\mathtt{ld})$ является результатом его компиляции), (iv) выполняет инструкцию чтения и (v) обещание $\langle z : 1@\_,\_\rangle$.

Для выражения запаздывания используется два отношения симуляции, $\mathcal{I}$ и $\mathcal{I}_{\text{pre}}$. Первое отношение не позволяет обещающей машине запаздывать слишком сильно: если состояния машин связаны посредством $\mathcal{I}$, то в каждом потоке обещающая машина полностью выполнила префикс программы, который выполнен ARM-машиной, и ожидает, пока ARM-машина исполнит следующую инструкцию. Второе отношение сигнализирует, что существует единственный поток, в котором обещающая машина может и должна “догнать” ARM-машину. Для этих отношений доказывается, что если состояния машин связаны отношением $\mathcal{I}_{\text{pre}}$, то существует конечная последовательность шагов обещающей машины, после выполнения которой состояния машин связаны отношением $\mathcal{I}$.

## 3.4 Формальное определение модели ARMv8 POP

В этом разделе приводится формализация модели ARMv8 POP, сделанная диссертантом, поскольку в [9] модель описана лишь словесно. Синтаксис программ, на которых определена модель, приведён на рис. 23. Программа представ-

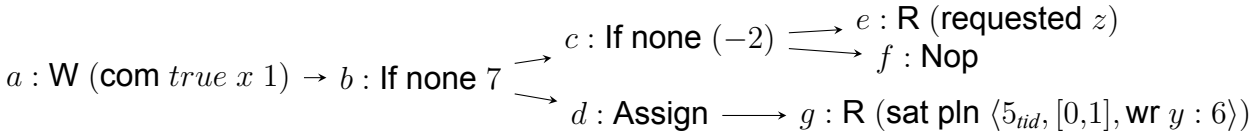



Рис. 24. Состояния подсистемы управления потока в модели ARMv8 POP

ляется как функция $Prog : Tid \to List\ S$, которая по идентификатору потока возвращает список его инструкций. Инструкции бывают следующих типов: чтение ($reg := [expr]$), запись ($[expr_0] := expr_1$), барьер памяти ($\mathsf{fence}(fmod_{\mathrm{ARM}})$), условный переход (if *expr* goto *k*), присваивание в локальную переменную ($reg := expr$), пустая операция (nop).

В этом языке опосредованно, через условные переходы назад, присутствуют циклы, поэтому каждая инструкция может быть исполнена несколько раз и в разных контекстах. Различные исполнения инструкции мы будем называть *экземплярами инструкций* (instruction instances) или просто экземплярами. Подсистема управления ARM-машины может исполнять экземпляры не по порядку и спекулятивно. Более того, исполнение экземпляра происходит за несколько шагов, которые могут быть разделены во времени. Как следствие, в каждый конкретный момент времени поток может находится в состоянии исполнения многих экземпляров.

Аналогично модели памяти Power [33], в представленной формализации используется граф типа дерево для описания состояния подсистемы управления. Пример такого графа приведён на рис. 24. Вершины дерева помечены состоянием экземпляров инструкций. Ребра дерева выражают программный порядок между экземплярами, где вершины с двумя исходящими ребрами представляют условные переходы, условие которых ещё не вычислено.

Каждый экземпляр инструкции идентифицируется парой ($tid$,$path$), где *tid* — это идентификатор потока, а $path : Path \triangleq List\ \mathbb{N}$ — путь в дереве подсистемы управления от его корня до экземпляра. Этот путь представлен как список позиций инструкций, соответствующих предшествующим экземплярам. Так, путь экземпляра $a$ на рис. 24 — это список $[0]$, путь экземпляра $b$ — $[0,1]$, а путь экземпляра $f$ — $[0,1,8,9]$.

Граф состояния подсистемы управления представлен в виде т.н. *плёнки* $tape : Tape \triangleq Path \rightharpoonup TapeCell$ — частичной функции, которая для пути экземпляра возвращает его состояние. Корректная плёнка является префикс-замкнутой:

$$
\begin{aligned}
\mathit{tapecell} \;::=\; & \mathsf{R}\ \mathit{st}_{\mathrm{read}} \mid \mathsf{W}\ \mathit{st}_{\mathrm{write}} \\
& \mid \mathsf{F}\ \mathit{st}_{\mathrm{fence}}\ \mathit{fmod}_{\mathrm{ARM}} \\
& \mid \mathsf{If}\ \mathit{st}_{\mathrm{ifgoto}}\ k \mid \mathsf{Assign} \mid \mathsf{Nop} \\
& : \mathit{TapeCell}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{sat\text{-}state} \;::=\; & \mathsf{pln} \mid \mathsf{inflight} \mid \mathsf{com} \\
\mathit{st}_{\mathrm{read}} \;::=\; & \mathsf{none} \mid \mathsf{requested}\ \ell \\
& \mid \mathsf{sat}\ \mathit{sat\text{-}state}\ \langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val} \rangle \\
\mathit{st}_{\mathrm{write}} \;::=\; & \mathsf{none} \\
& \mid \mathsf{pending}\ \ell\ \mathit{val} \\
& \mid \mathsf{com}\ \mathit{bool}\ \ell\ \mathit{val} \\
\mathit{st}_{\mathrm{fence}} \;::=\; & \mathsf{none} \mid \mathsf{com} \\
\mathit{st}_{\mathrm{ifgoto}} \;::=\; & \mathsf{none} \mid \mathsf{taken} \mid \mathsf{ignored} \\
\mathit{val} : \mathit{Val} \;=\; & \mathit{Loc} \cup \mathbb{Z}
\end{aligned}
$$

Рис. 25. Язык состояния исполнения инструкций в ARMv8 POP

если она определена для некоторого пути, то она определена и для префикса этого пути.

Состояние экземпляра представляется с помощью *ячейки плёнки* $\mathit{tapecell} : \mathit{TapeCell}$ (см. рис. 25). Её синтаксис симметричен синтаксису инструкций. Рассмотрим возможные состояния экземпляра в зависимости от его типа.

Экземпляр инструкции чтения $\mathit{st}_{\mathrm{read}}$ может находиться в одном из трёх состояний: (i) $\mathsf{none}$ — экземпляр только загружен или его исполнение перезапущено; (ii) $\mathsf{requested}\ \ell$ — экземпляр отправил запрос в подсистему памяти; (iii) $\mathsf{sat}\ \mathit{sat\text{-}state}\ \langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val} \rangle$ — экземпляра получил ответ от экземпляра записи $(\mathit{tid}, \mathit{path})$ со значением $\mathit{val}$. Поле *sat-state* сигнализирует об одной из трёх ситуаций: (i) чтение произведено из экземпляра записи, исполнение которого ещё не завершено ($\mathsf{inflight}$); (ii) чтение произведено из завершенного экземпляра записи ($\mathsf{pln}$); или (iii) чтение произведено из завершенного экземпляра записи, и исполнение самого экземпляра чтения завершено ($\mathsf{com}$).

Экземпляр инструкции записи $\mathit{st}_{\mathrm{write}}$ может быть также в одном из трёх состояний: (i) $\mathsf{none}$; (ii) $\mathsf{pending}\ \ell\ \mathit{val}$ — целевые адрес и значение экземпляра определены, и экземпляры чтения из того же потока могут читать из этого экземпляра записи; (iii) $\mathsf{com}\ \mathit{bool}\ \ell\ \mathit{val}$ — исполнение экземпляра завершено, и если

| $i$ | $cmds[i]$ | *path* | *tape*(*path*) |
|---|---|---|---|
| 0 | $a := [x];$ | 0 | R none |
| 1 | $[y] := a;$ | 0,1 | W none |
| 2 | $a := [z];$ | 0,1,2 | R (sat pln $\langle 8_{tid}, [0,1,2,3], \mathsf{wr}\ z : 9\rangle$) |
| 3 | $[w] := a;$ | 0,1,2,3 | W none |

Табл. 3. Пример состояния подсистемы управления модели ARMv8 POP

$bool = true$, то соответствующий запрос отправлен в подсистему памяти, иначе экземпляр может быть прочитан только экземплярами чтения того же потока.

Исполнение экземпляра барьера памяти $st_{\text{fence}}$ может быть либо завершено (com), либо нет (none). Состояние экземпляра условного перехода $st_{\text{ifgoto}}$ указывает, что (i) либо условие было вычислено к не нулевому значению, и переход на *k* позиций в списке инструкций произошёл (taken); (ii) либо — к нулевому значению, и переход произошёл к следующей инструкции (ignored); (iii) либо выбор ещё не сделан (none). Экземпляры присваивания и пустой операции только загружаются и не исполняются, поэтому их состояния не имеют параметров.

Стоит отметить, что поскольку в каждый момент несколько экземпляров (разных) инструкций могут исполняться в одном потоке, то невозможно определить конкретное значение локальной переменной — оно может варьироваться в контексте разных экземпляров.

Рассмотрим пример плёнки (см. табл. 3). Здесь программа состоит из четырёх инструкций (столбец $cmds[i]$), каждая из которых имеет по запущенному экземпляру (столбец *tape*(*path*)). При этом экземпляр инструкции чтения $a := [z]$ получил ответ со значением 9, в то время как экземпляр $a := [x]$ ещё не отправил запрос. Как следствие, переменная $a$ имеет значение 9 на пути $[0,1,2,3]$ (после экземпляра чтения $a := [z]$) и не определена на других путях.

Введём функции $\mathsf{regf}, \mathsf{regf}_{\mathsf{com}} : (\mathit{List}\ S \times \mathit{Tape} \times \mathit{Path}) \to (\mathit{Reg} \rightharpoonup \mathit{Val})$, где $\mathsf{regf}(cmds,\mathit{tape},\mathit{path})$ и $\mathsf{regf}_{\mathsf{com}}(cmds,\mathit{tape},\mathit{path})$ представляют состояние локальных переменных, актуальное для экземпляра с путём *path*, т.е. сразу “перед” его исполнением (см. рис. 26). Отличие между функциями, которое выделено серым на рис. 26, заключается в том, что regf учитывает все экземпляры чтения, на которые получены ответы, а $\mathsf{regf}_{\mathsf{com}}$ — только завершённые (committed). Значения функций regf и $\mathsf{regf}_{\mathsf{com}}$ для примера из табл. 3 приведены в табл. 4.

$\mathsf{regf}(cmds, \mathit{tape}, []) = \mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, []) = \bot;$

$\forall i.\ \mathsf{regf}(cmds, \mathit{tape}, [i]) = \mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, [i]) = \bot;$

$\forall i, j.\ cmds[i] =$ "$\mathit{reg}\ :=\ [\mathit{expr}]$" $\Rightarrow$

$\mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i)[\mathit{reg} \mapsto m] \wedge$

$\mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i)[\mathit{reg} \mapsto m_{com}]$, где

$m = \mathit{val}, m_{com} = \mathit{val},$

если $\mathit{tape}(\mathit{path}) = \mathsf{R}$ sat com $\langle \_, \_, \mathsf{wr}\ \_ : \mathit{val} \rangle$;

$m = \mathit{val}, m_{com} = \bot,$

если $\mathit{tape}(\mathit{path}) = \mathsf{R}$ sat *sat-state* $\langle \_, \_, \mathsf{wr}\ \_ : \mathit{val} \rangle$, *sat-state* $\neq$ com;

$m = \bot, m_{com} = \bot,$

если $\mathit{tape}(\mathit{path}) = \mathsf{R}\ st_{\text{read}}, st_{\text{read}} \in \{\mathsf{none}, \mathsf{requested}\ \_\}$;

$\forall i, j.\ cmds[i] =$ "$\mathit{reg}\ :=\ \mathit{expr}$" $\Rightarrow$

$\mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i)[a \mapsto [\![\mathit{expr}]\!]^{\mathit{path}:i}] \wedge$

$\mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i)[a \mapsto [\![\mathit{expr}]\!]^{\mathit{path}:i}_{\mathsf{com}}];$

$\forall i, j.\ \mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}\quad(cmds, \mathit{tape}, \mathit{path} : i) \wedge$

$\mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i : j) = \mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path} : i)$, иначе.

Рис. 26. Функции вычисления состояния переменных $\mathsf{regf}$ и $\mathsf{regf}_{\mathsf{com}}$

| *path* | $\mathsf{regf}(cmds, \mathit{tape}, \mathit{path})$ | $\mathsf{regf}_{\mathsf{com}}(cmds, \mathit{tape}, \mathit{path})$ |
|---|---|---|
| 0 | $\bot$ | $\bot$ |
| 0,1 | $\bot$ | $\bot$ |
| 0,1,2 | $\bot$ | $\bot$ |
| 0,1,2,3 | $[a \mapsto 9]$ | $\bot$ |

Табл. 4. Значение функций $\mathsf{regf}$ и $\mathsf{regf}_{\mathsf{com}}$ для примера из табл. 3

Функции значения переменных естественным образом обобщаются на функции вычисления значения выражений $\mathsf{e} \rightharpoonup \mathit{Val}$. Когда значения параметров $cmds$, *tape* (и *path*) очевидны из контекста, мы обозначаем эти функции $[\![-]\!]^{path}$ и $[\![-]\!]^{path}_{\mathsf{com}}$ (или $[\![-]\!]$ и $[\![-]\!]_{\mathsf{com}}$).

Состояние подсистемы памяти $M_{\text{POP}}$ является тройкой $\langle \mathit{Evt}, \mathit{Ord}, \mathit{Prop} \rangle$, где $\mathit{Evt} \subseteq \mathit{ReqSet}$ — это множество запросов в подсистеме, $\mathit{Ord} \subseteq \mathit{Evt} \times \mathit{Evt}$ — частичный порядок на запросах и $\mathit{Prop} \subseteq \mathit{Tid} \times \mathit{Evt}$ — функция, которая по идентификатору потока возвращает множество запросов, о которых он осведомлён. Запрос *req* является тройкой $\langle \mathit{tid}, \mathit{path}, \mathit{reqinfo} \rangle$, где *tid* — идентификатор потока, который отправил это запрос, *path* — путь до соответствующего экземпляра инструкции и *reqinfo* — информация о типе и параметрах запроса:

$$\mathit{reqinfo} ::= \mathsf{rd}\ \ell \mid \mathsf{wr}\ \ell : \mathit{val} \mid \mathsf{dmb}.$$

Запрос чтения параметризован целевой локацией, запрос записи — целевой локацией и записываемым значением.

Полное состояние ARM-машины $\mathsf{State}_{\text{ARM}}$ — это тройка $\langle M_{\text{POP}}, \mathit{iordf}, \mathit{tapef} \rangle$, где $M_{\text{POP}}$ — это состояние подсистемы памяти, $\mathit{iordf} : \mathit{Tid} \to \mathit{List}\ \mathit{ReqSet}$ — это функция, которая для потока возвращает упорядоченный список отправленных им запросов чтения и $\mathit{tapef} : \mathit{Tid} \to \mathit{Tape}$ представляет плёнки потоков (состояние подсистемы управления). Компонента *iordf* используется для разрешения конфликтов, вызванных отправлением запросов чтения к одной локации не по порядку (подробнее описано ниже в правиле **Получение ответа на чтение**).

Начальное состояние ARM-машины имеет инициализирующие записи во все локации $\mathit{Evt}^{\text{init}} \triangleq \{\langle 0_{\mathit{tid}}, [], \mathsf{wr}\ \ell : 0 \rangle \mid \ell \in \mathit{Loc}\}$, где соответствующие запросы не упорядочены и о них осведомлены все потоки:

$$\mathbf{s}^{\text{init}} \triangleq \langle M_{\text{POP}} = \langle \mathit{Evt}^{\text{init}}, \varnothing, \mathit{Tid} \times \mathit{Evt}^{\text{init}} \rangle, \mathit{iordf} = \lambda \mathit{tid}.\ [], \mathit{tapef} = \lambda \mathit{tid}.\ \bot \rangle.$$

Наша формализация ARM-машины имеет 12 правил перехода. Эти переходы в математической нотации приведены в приложении Б, ниже приведено их неформальное описание.

**Загрузка инструкции** *tid path* (fetch instruction) добавляет экземпляр инструкции с состоянием $\mathsf{none}$ в плёнку *tape* потока *tid*.

**Уведомление потока** *tid* **о запросе** $e$ (propagate) добавляет $e$ во множество $M_{\text{POP}}.\mathit{Prop}(\mathit{tid})$. Правило проверяет, что поток уведомлён о всех запросах $e'$ таких, что $e' <_{\mathit{Ord}} e$. Правило также добавляет по ребру $(e, e'')$ в *Ord*

для каждого запроса $e''$, который не переупорядочиваем с $e$ и о котором осведомлён поток $e.tid$, но не поток *tid*.

**Выбор ветки условного перехода** *tid path* (branch commit) выполняет экземпляр инструкции условного перехода $\mathsf{if} - \mathsf{goto}$ и удаляет одну из веток спекулятивного исполнения из плёнки потока *tid* вместе с соответствующими ей запросами из подсистемы памяти.

**Завершение барьера** `ld` *tid path* (fence commit) проверяет, что предшествующие экземпляры чтения завершены (committed), и помечает экземпляр барьера как выполненный.

**Завершение барьера** `sy` *tid path* проверяет, что все предшествующие экземпляры завершены, помечает экземпляр барьера как выполненный и отправляет запрос барьера в подсистему памяти.

**Подготовка записи** *tid path* $\ell$ *val* (write pending) устанавливает состояние экземпляра записи в $\mathsf{pending}\ \ell\ val$, где $\ell$ и *val* являются целевой локацией и записываемым значением, которые вычисляются в коде инструкции.

**Завершение записи** *tid path* $\ell$ *val* (write commit) устанавливает состояние экземпляра записи в $\mathsf{com}\ \_\ \ell\ val$. Это правило также отправляет соответствующий запрос в подсистему памяти, но только в том случае, если в плёнке потока нет последующего экземпляра записи в ту же локацию, исполнение которого завершено. Кроме того, правило перезапускает некоторые исполнения экземпляров чтения, целевой локацией которых является $\ell$, а также экземпляры произвольных типов, зависящие от этих экземпляров чтения. Правило имеет несколько ограничений. Так, исполнение всех экземпляров барьеров памяти и условных переходов, предшествующих экземпляру $(tid, path)$, должно быть завершено; все предшествующие экземпляры должны иметь полностью вычисленные целевые локации (fully determined addresses), т.е. соответствующие выражения адресов должны быть вычисляемы к значению с помощью функции $\mathsf{regf}_{\mathsf{com}}$.

**Отправка запроса на чтение** *tid path* $\ell$ (read issue) добавляет соответствующий запрос в подсистему хранения, а также в список отправленных запросов (компонента $iordf(tid)$). Правило требует, чтобы исполнение предшествующих экземпляров барьеров было завершено.

**Получение ответа на чтение** *tid path tid′ path′* $\ell$ *val* (read satisfy) и **Игнорирование ответа на чтение** *tid path tid′ path′* $\ell$ *val* (read satisfy fail) получает ответ на запрос $\langle tid, path, \mathsf{rd}\ \ell\rangle$ из запроса записи $\langle tid', path', \mathsf{wr}\ \ell\ :$

$val\rangle$, если между ними нет запроса в отношении *Ord*. Правила переходов удаляют запрос на чтение из подсистемы памяти. Если в плёнке потока не присутствует предшествующего экземпляра чтения из той же локации, которой отправил запрос в подсистему памяти после экземпляра $(tid, path)$ (что определяется с помощью компоненты $iordf(tid)$) и получил ответ от другого запроса записи, то применяется переход **Получение ответа на чтение**. Этот переход присваивает экземпляру чтения состояние $\mathsf{sat\ pln}\ \langle tid', path', \mathsf{wr}\ \ell : val\rangle$ и перезапускает некоторые последующие экземпляры чтения из той же локации вместе с зависимостями, если это требуется для восстановления корректности. Если условие не выполняется, то применяется переход **Игнорирование ответа на чтение**, который перезапускает экземпляр $(tid, path)$.

**Получение ответа на чтение от подготовленной записи** *tid path path′ ℓ val* (read satisfy from in-flight write) устанавливает состояние экземпляра чтения в $\mathsf{sat\ inflight}\ \langle tid, path', \mathsf{wr}\ \ell : val\rangle$ при условии, что существует подготовленный незавершенный экземпляр записи $(tid, path')$ и не существует экземпляра записи в ту же локацию между ними или экземпляра чтения из той же локации, который получил ответ от другой записи. Это правило также, как и правило **Получение ответа на чтение**, перезапускает некоторые последующие экземпляры чтения.

**Завершение чтения** *tid path* (read commit) проверяет, что все предшествующие экземпляры барьеров памяти и условных переходов завершены, все предшествующие экземпляры имеют полностью определённые адреса, и присваивает экземпляру состояние $\mathsf{sat\ com}\ \_$.

## 3.5 Формальное определение обещающей модели

Состояние обещающей машины $\mathsf{State}_{\mathsf{Promise}}$ является парой $\langle \mathcal{TS}, M\rangle$. При этом $M \subset Msg$ — это память машины, являющаяся множеством сообщений вида $\langle \ell : val@\tau, view\rangle : Msg$, состоящих из целевой локации $\ell : Loc$, значения $val : Val$, метки времени $\tau : \mathtt{Time} = \mathbb{Q}$, и фронта сообщения $view : \mathcal{V} = Loc \to \mathtt{Time}$. $\mathcal{TS} : Tid \to \mathsf{TS}$ — это функция, которая по идентификатору потока возвращает его состояние. Состояние потока $TS : \mathsf{TS}$ является тройкой $\langle \sigma, V, promises\rangle$, где $\sigma$ — это локальное состояние потока в рамках описанной ниже помеченной системы переходов (labeled transition system, LTS), $V = \langle view_{\text{cur}}, view_{\text{acq}}, view_{\text{rel}}\rangle : \mathcal{V} \times \mathcal{V} \times \mathcal{V}$

(READ-HELPER)

$$\frac{\mathsf{cur}(\ell) \leqslant t \qquad \mathsf{cur}' = \mathsf{cur} \sqcup [\ell@\tau] \qquad \mathsf{acq}' = \mathsf{acq} \sqcup \mathsf{cur}'}{\langle \mathsf{cur},\mathsf{acq},\mathsf{rel}\rangle \xrightarrow{\mathtt{R}(\ell,\tau,R)} \langle \mathsf{cur}',\mathsf{acq}',\mathsf{rel}\rangle}$$

(WRITE-HELPER)

$$\frac{\mathsf{cur}(\ell) < t \qquad R = \mathsf{rel} \sqcup [\ell@\tau] \qquad \mathsf{cur}' = \mathsf{cur} \sqcup [\ell@\tau] \qquad \mathsf{acq}' = \mathsf{acq} \sqcup \mathsf{cur}'}{\langle \mathsf{cur},\mathsf{acq},\mathsf{rel}\rangle \xrightarrow{\mathtt{W}(\ell,\tau,R)} \langle \mathsf{cur}',\mathsf{acq}',\mathsf{rel}\rangle}$$

(READ)

$$\frac{\langle \ell : v@\tau,R\rangle \in M \qquad \sigma \xrightarrow{\mathtt{R}(\ell,v)} \sigma' \qquad \mathcal{V} \xrightarrow{\mathtt{R}(\ell,\tau,R)} \mathcal{V}'}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}', P\rangle, M\rangle}$$

(WRITE)

$$\frac{\sigma \xrightarrow{\mathtt{W}(\ell,v)} \sigma' \qquad \mathcal{V} \xrightarrow{\mathtt{W}(\ell,\tau,R)} \mathcal{V}' \qquad m = \langle \ell : v@\tau,R\rangle \qquad \forall v', \mathit{view}'.\ \langle \ell : v'@\tau,\mathit{view}'\rangle \notin M \qquad M' = M \cup \{m\} \qquad P' = P \setminus \{m\}}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}', P'\rangle, M'\rangle}$$

(ACQ-FENCE)

$$\frac{\sigma \xrightarrow{\mathtt{F}(\mathtt{acq})} \sigma'}{\langle\langle \sigma, \langle \mathsf{cur},\mathsf{acq},\mathsf{rel}\rangle, P\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \langle \mathsf{acq},\mathsf{acq},\mathsf{rel}\rangle, P\rangle, M\rangle}$$

(REL-FENCE)

$$\frac{\sigma \xrightarrow{\mathtt{F}(\mathtt{rel})} \sigma'}{\langle\langle \sigma, \langle \mathsf{cur},\mathsf{acq},\mathsf{rel}\rangle, \varnothing\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \langle \mathsf{cur},\mathsf{acq},\mathsf{cur}\rangle, \varnothing\rangle, M\rangle}$$

(SILENT)

$$\frac{\sigma \xrightarrow{\varepsilon} \sigma'}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma', \mathcal{V}, P\rangle, M\rangle}$$

(PROMISE)

$$\frac{\forall v, \mathit{view}.\ \langle m.\ell : v@m.\tau,\mathit{view}\rangle \notin M \qquad P' = P \cup \{m\} \qquad M' = M \cup \{m\} \qquad \forall \ell.\ \exists v, \mathit{view}.\ \langle \ell : v@m.\mathtt{view}(\ell),\mathit{view}\rangle \in M'}{\langle\langle \sigma, \mathcal{V}, P\rangle, M\rangle \xrightarrow[\text{Promise } tid]{} \langle\langle \sigma, \mathcal{V}, P'\rangle, M'\rangle}$$

(GLOBAL)

$$\frac{\langle \mathit{TS}, M\rangle \xrightarrow[\text{Promise } \mathit{tid}]{} \langle \mathit{TS}', M'\rangle \qquad \langle \mathit{TS}', M'\rangle \xrightarrow[\text{Promise } \mathit{tid}]{}{}^{*} \langle \mathit{TS}'', M''\rangle \qquad \mathit{TS}''.\mathit{promises} = \varnothing}{\langle \mathcal{TS}[\mathit{tid} \mapsto \mathit{TS}], M\rangle \xrightarrow[\text{Promise}]{} \langle \mathcal{TS}[\mathit{tid} \mapsto \mathit{TS}'], M'\rangle}$$

Рис. 27. Переходы обещающей машины

— это базовый, приобретающий и высвобождающий фронты потока, $\mathit{promises} \subset \mathit{Msg}$ — это множество сообщений, которые были обещаны потоком, но еще не выполнены.

Обещающая модель памяти задана на помеченной системе переходов, а не на программах непосредственно. В приложении Г.1 приводится метод построения системы переходов по программе. Метка в системе переходов может быть: (i) операцией чтения из локации $\ell$ значения $v$ ($\mathtt{R}(\ell,v)$), (ii) операцией записи в локацию $\ell$ значения $v$ ($\mathtt{W}(\ell,v)$), (iii) барьером памяти с модификатором *fmod* ($\mathtt{F}(\mathit{fmod})$), (iv) внутренним переходом ($\varepsilon$). Последний тип описывает действия потока, кото-

рые не затрагивают память, например, присваивание в локальную переменную и исполнение оператора условного перехода.

Стоит отметить, что в работе [23], на основе которой написана эта глава, обещающая модель задана непосредственно на синтаксисе модели ARMv8 POP. Тем не менее, для того, чтобы не вводить разные определения в главах 3 и 4, в диссертации обещающая модель определяется “поверх” системы переходов. Также в доказательстве корректности компиляции автор диссертации исходит из предположения, что система переходов может быть восстановлена по программе согласовано со схемой компиляции и функциями вычисления значения модели ARMv8 POP (см. приложения Г.1, Г.2 и Г.3).

Перед исполнением любой программы память обещающей машины состоит из записей во все локации $M^{\text{init}} \triangleq \{\langle \ell : 0@\mathbf{0}, \mathit{view}^{\text{init}} \rangle \mid \ell \in \mathit{Loc}\}$, где $\mathit{view}^{\text{init}} \triangleq \lambda \ell.\ \mathbf{0}$. В целом начальное состояние машины выглядит так:

$$\mathbf{p}^{\text{init}} \triangleq \langle \lambda \mathit{tid}.\ \langle \sigma = \sigma^{\text{init}}, V = \langle \mathit{view}^{\text{init}}, \mathit{view}^{\text{init}}, \mathit{view}^{\text{init}} \rangle, \mathit{promises} = \varnothing \rangle, M^{\text{init}} \rangle.$$

Перейдем к описанию шагов исполнения обещающей машины (см. рис. 27). Только правило (GLOBAL) оперирует над всем состоянии машины, остальные правила заданы локально для потока. Правило (GLOBAL) требует от потока, который совершает локальный переход, провести *сертификацию* обещаний, т.е. показать, что существует изолированный запуск потока, в рамках которого он выполнит все свои обещания.

**Выполнение приобретающего барьера** (ACQ-FENCE) делает базовый фронт потока $\mathit{view}_{\text{cur}}$, равным приобретающему фронту потока $\mathit{view}_{\text{acq}}$, где последний является объединением фронтов сообщений, прочитанных потоком к этому моменту, и меток времени записей, произведенных потоком.

**Выполнение высвобождающего барьера** (REL-FENCE) делает высвобождающий фронт потока $\mathit{view}_{\text{rel}}$ равным базовому фронту потока $\mathit{view}_{\text{cur}}$. В результате фронты сообщений, которые поток добавит в память после выполнения высвобождающего барьера, будут содержать информацию о записях, совершённых потоком до выполнения высвобождающего барьера. Кроме того, переход, соответствующий высвобождающему барьеру, имеет дополнительное ограничение — в момент его выполнения у потока не должно быть невыполненных обещаний, т.е. *promises* должен быть равен пустому множеству.

**Чтение из локации** $\ell$ (READ) выполняется потоком следующим образом. Поток выбирает из памяти машины некоторое сообщение $\langle \ell : val@\tau,view \rangle$. При этом метка времени записи $\tau$ должна быть больше, чем значение базового фронта потока $view_{\text{cur}}(\ell)$, а само сообщение не должно находится во множестве обещанных, но не выполненых потоком сообщений *promises*. Это правило увеличивает базовый фронт потока на $[\ell@\tau]$, а приобретающий — на *view*.

**Обещание записи** $\langle \ell : val@\tau,view \rangle$ (PROMISE) добавляет сообщение в память машины и в множество *promises*. При этом целевая локация $\ell$ и значение *val* могут быть произвольными, т.е. они не зависят от локального состояния потока $\sigma$. Метка времени должна быть уникальной среди сообщений локации $\ell$, уже находящихся в памяти. Этот переход не обновляет фронты потока. Также легко заметить, что этот переход недетерминирован, но ограничен сертификацией.

**Выполнение обещания** $\langle \ell : val@\tau,view \rangle$ (FULFILL) удаляет сообщение из множества невыполненных обещаний *promises*. При этом данный переход должен быть возможен в рамках помеченной системы переходов локальных состояний, т.е. должно существовать состояние $\sigma'$ такое, что $\sigma \xrightarrow{\mathbb{W}(\ell,val)} \sigma'$. При этом метка времени $\tau$ должна быть больше значения базового фронта $view_{\text{cur}}(\ell)$, и фронт сообщения *view* должен быть равен композиции высвобождающего фронта $view_{\text{rel}}$ и $[\ell@\tau]$. Переход также увеличивает базовый и приобретающий фронты потока на $[\ell@\tau]$.

**Внутренний переход** (SILENT) соответствует шагу исполнения потока, который не взаимодействует с памятью, например: присваивание в локальную переменную, исполнение условного перехода или пустой операции. Такой переход меняет только локальное состояние потока.

## 3.6 Промежуточная машина ARM+$\tau$

В этом разделе описывается промежуточная машина ARM+$\tau$ , а также доказывается, что она симулирует ARM-машину. Для этого сначала показывается корректность некоторых базовых утверждений про подсистему памяти ARM-машины, а затем приводится подход для ввода меток времени для конкретного сценария поведения ARMv8 POP.

### 3.6.1 Свойства подсистемы памяти модели ARMv8 POP

**Лемма 1.** Пусть состояние ARM-машины **s** достижимо из начального. Тогда отношение **s**.*Ord* равно транзитивному замыканию разности его самого и отношения $\hookrightarrow$. Кроме того, оно ациклично.

$$\forall \mathbf{s}.\ \mathit{Prog} \vdash \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s} \Rightarrow \mathbf{s}.\mathit{Ord} = (\mathbf{s}.\mathit{Ord} \setminus \hookrightarrow)^{+} \wedge (\mathbf{s}.\mathit{Ord} \text{ ациклично}).$$

*Доказательство.* Утверждение верно для начального состояния $\mathbf{s}^{\text{init}}$. Изменения в подсистеме памяти бывают трёх типов: (i) удаление запроса, (ii) отправка запроса в подсистему и (iii) уведомление потока о запросе. Рассмотрим, как меняется состояние подсистемы памяти при каждом из этих изменений в предположении, что начальное состояние памяти равно $\langle \mathit{Evt}, \mathit{Ord}, \mathit{Prop} \rangle$, а изменённое — $\langle \mathit{Evt}', \mathit{Ord}', \mathit{Prop}' \rangle$:

**Удаление запроса** $e$

$$\begin{aligned}
&\mathit{Evt}' = \mathit{Evt} \setminus \{e\},\\
&\mathit{Prop}' = \mathit{Prop} \setminus \{(\mathit{tid}, e) \mid \mathit{tid}\},\\
&\mathit{Ord}' = (\mathit{Ord} \setminus (\{e\} \times \mathit{Evt}) \setminus (\mathit{Evt} \times \{e\}) \setminus \hookrightarrow)^{+}.
\end{aligned}$$

**Отправка запроса** $e$ **потоком** *tid*

$$\begin{aligned}
&\mathit{Evt}' = \mathit{Evt} \cup \{e\},\\
&\mathit{Prop}' = \mathit{Prop} \cup \{(\mathit{tid}, e)\},\\
&\mathit{Ord}' = (\mathit{Ord} \cup \{(e', e) \mid \mathit{Prop}(\mathit{tid}, e'), e' \not\hookrightarrow e\})^{+}.
\end{aligned}$$

**Поток** *tid* **становится осведомлённым о запросе** $e$

$$\begin{aligned}
&\mathit{Evt}' = \mathit{Evt},\\
&\mathit{Prop}' = \mathit{Prop} \cup \{(\mathit{tid}, e)\},\\
&\mathit{Ord}' = (\mathit{Ord} \cup \{(e, e') \mid \mathit{Prop}(\mathit{tid},e'), \neg\mathit{Prop}(e.\mathit{tid},e'), e \not\hookrightarrow e', \neg(e' <_{\mathit{Ord}} e)\})^{+}.
\end{aligned}$$

Очевидно, что во всех трёх случаях $\mathit{Ord}' = (\mathit{Ord}' \setminus \hookrightarrow)^{+}$. Также отношение $\mathit{Ord}'$ ациклично, поскольку в случае удаления события $\mathit{Ord}' \subseteq \mathit{Ord}$, отправка запроса добавляет в *Ord*-компоненту только ребра, входящие в новый запрос $e$, а в третьем случае происходит проверка на ацикличность $\mathit{Ord}'$. □

Леммы 2 и 3 доказываются аналогично.

**Лемма 2.** Пусть состояние ARM-машины $\mathbf{s}$ достижимо из начального. Тогда для любых двух запросов, которые не могут быть переупорядочены и о которых осведомлён один и тот же поток, верно, что запросы совпадают или между ними есть ребро в отношении $\mathbf{s}.Ord$.

$$\forall \mathbf{s}, e, e', tid.\ Prog \vdash \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s} \wedge e \not\leftrightarrow e' \wedge \mathbf{s}.Prop(tid, e) \wedge \mathbf{s}.Prop(tid, e') \Rightarrow \\ e = e' \vee \mathbf{s}.Ord(e, e') \vee \mathbf{s}.Ord(e', e).$$

**Лемма 3.** Пусть состояние ARM-машины $\mathbf{s}'$ достижимо из состояния $\mathbf{s}$. Тогда все запросы записи и барьеров памяти, которые находятся в подсистеме памяти состояния $\mathbf{s}$, также находятся в подсистеме памяти состояния $\mathbf{s}'$.

$$\forall \mathbf{s}, \mathbf{s}'.\ Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow \mathbf{s}.Evt \setminus \{e \mid e \text{ — запрос чтения}\} \subseteq \mathbf{s}'.Evt.$$

**Лемма 4.** Пусть состояние ARM-машины $\mathbf{s}'$ достижимо из состояния $\mathbf{s}$. Тогда рёбра отношения $\mathbf{s}.Ord$, которые связывают запросы из множества $\mathbf{s}'.Evt$, также являются рёбрами в отношении $\mathbf{s}'.Ord$.

$$\forall \mathbf{s}, \mathbf{s}'.\ Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow \mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt) \subseteq \mathbf{s}'.Ord.$$

*Доказательство.* Следующая, более слабая, версия утверждения верна, так как нет перехода, который бы удалял *Ord*-ребро между запросами, который не могут быть переупорядочены:

$$\forall \mathbf{s}, \mathbf{s}'.\ Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{s}' \Rightarrow (\mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt)) \setminus \hookrightarrow \subseteq \mathbf{s}'.Ord$$

Теперь докажем исходное утверждение. Зафиксируем пару запросов $(e, e') \in \mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt)$. Мы хотим показать, что $(e, e') \in \mathbf{s}'.Ord$. Если запросы не переупорядочиваемы, $e \not\leftrightarrow e'$, то утверждение выполняется, т.к. $(e, e') \in (\mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt)) \setminus \hookrightarrow \subseteq \mathbf{s}'.Ord$. Иначе $e$ и $e'$ являются запросами чтения или записи к разным локациям. Поскольку $\mathbf{s}.Ord(e, e')$, то по лемме 1 существует конечный путь по $\mathbf{s}.Ord$-рёбрам от $e$ к $e'$, что каждое ребро этого пути соединяет не переупорядочиваемые запросы.

Предположим, что на этом пути есть запрос барьера памяти $e''$. Тогда из транзитивности $\mathbf{s}.Ord$ следует, что $\{(e, e''), (e'', e')\} \subseteq \mathbf{s}.Ord$. Следовательно, по ослабленной версии леммы и транзитивности $\mathbf{s}'.Ord$ следует, что $\{(e, e''), (e'', e'), (e, e')\} \subseteq \mathbf{s}'.Ord$.

Теперь предположим, что на этом пути нет запроса барьера памяти. Тогда по определению отношения $\hookrightarrow$ следует, что все запросы на этом пути оперируют одной и той же локацией. Это противоречит $e \not\leftrightarrow e'$. □

### 3.6.2 Модель ARMv8 POP и метки времени

Зафиксируем некоторую программу *Prog* и рассмотрим её завершающийся сценарий поведения в модели ARMv8 POP:

$$\mathit{Prog} \vdash \mathbf{s}^0 \xrightarrow[\text{ARM}]{} \mathbf{s}^1 \xrightarrow[\text{ARM}]{} \ldots \xrightarrow[\text{ARM}]{} \mathbf{s}^n,$$

где $\mathbf{s}^0 = \mathbf{s}^{\text{init}}$ — это начальное состояние машины; $\mathbf{s}^n = \mathbf{s}$ является финальным состоянием, т.е. в подсистеме памяти $\mathbf{s}$ нет запросов чтения, все потоки осведомлены о всех запросах, исполнение всех экземпляров инструкций завершено, и загрузка новых экземпляров невозможна.

Для локации $\ell$ и множества запросов *Evt* мы определяем $\mathit{Evt}_\ell$ как множество всех запросов записи из *Evt*, целевой локацией которых является $\ell$:

$$\mathit{Evt}_\ell \triangleq \{\langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val}\rangle \in \mathit{Evt} \mid \mathit{tid}, \mathit{path}, \mathit{val}\}.$$

Поскольку не существует перехода, который удалял бы запросы записи, то $\mathbf{s}.\mathit{Evt}_\ell$ является множеством всех запросов записи в $\ell$, которые были отправлены в подсистему памяти за весь сценарий поведения.

Зафиксируем локацию $\ell$. Известно, что каждый поток осведомлён о каждом запросе $e \in \mathbf{s}.\mathit{Evt}_\ell$, т.к. состояние $\mathbf{s}$ является финальным. Также известно, что два запроса записи в одну локацию не переупорядочиваемы. Как следствие, по лемме 2 верно, что $mo_\ell \triangleq \mathbf{s}.\mathit{Ord}{\restriction_{\mathbf{s}.\mathit{Evt}_\ell}}$, где $R{\restriction_S} \triangleq R \cap (S \times S)$, является полным порядком на запросах записи в локацию $\ell$. Мы определяем отношение $mo \triangleq \bigcup_\ell mo_\ell$ как объединение множеств $mo_\ell$ по всем возможным локациям. Используя позиции запросов во множестве $mo_\ell$, мы определяем функцию, которая по запросу записи возвращает его метку времени:

$$\mathsf{map}_\tau(e) \triangleq \begin{cases} \mathsf{index}(mo_\ell, e), & \text{если } e = \langle \mathit{tid}, \mathit{path}, \mathsf{wr}\ \ell : \mathit{val}\rangle \in \mathbf{s}.\mathit{Evt}; \\ \bot, & \text{иначе.} \end{cases}$$

Наконец, мы показываем, что для любого состояния $\mathbf{s}^i$ из рассмотренного сценария поведения верно, что отношение $mo{\restriction_{\mathbf{s}^i.\mathit{Evt}}} \cup \mathbf{s}^i.\mathit{Ord}$ ациклично.

**Теорема 2.** $\forall i \leqslant n, mo{\restriction_{\mathbf{s}^i.\mathit{Evt}}} \cup \mathbf{s}^i.\mathit{Ord}$ ациклично.

*Доказательство.* Очевидно, что утверждение выполняется при $i = 0$. Предположим, что существует такой индекс $j$, что для любого индекса $i < j$ отношение

$mo{\restriction_{\mathbf{s}^i.Evt}} \cup \mathbf{s}^i.Ord$ ациклично, но отношение $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$ имеет цикл. Известно, что $mo{\restriction_{\mathbf{s}^n.Evt}} \cup \mathbf{s}^n.Ord = \mathbf{s}^n.Ord$ не имеет циклов. Тогда цикл из $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$ должен быть "удалён" на части сценария поведения $Prog \vdash \mathbf{s}^j \xrightarrow[\text{ARM}]{}{}^* \mathbf{s}^n$.

С этого момента мы будем различать *Ord*- и $mo$-ребра. Мы будем называть пару запросов $(e, e')$ ***Ord*-ребром**, если $(e, e') \in \mathbf{s}^j.Ord$, и $mo$***-ребром***, если $(e, e') \in mo{\restriction_{\mathbf{s}^j.Evt}} \setminus \mathbf{s}^j.Ord$.

Рассмотрим цикл минимальной длины в $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$. В этом цикле должно присутствовать $mo$-ребро, т.к. отношение $\mathbf{s}^j.Ord$ ациклично. Такое $mo$-ребро $(e, e')$ соединяет два запроса записи в локацию $\ell$, и при этом $\mathsf{map}_\tau(e) < \mathsf{map}_\tau(e')$. Поскольку это ребро является частью цикла, то существует путь от $e'$ к $e$, состоящий из *Ord*- и $mo$-рёбер. Мы можем разбить этот путь на максимальные гомогенные $mo$- и *Ord*-подпути. Покажем, что каждый *Ord*-подпуть состоит только из одного ребра.

Рассмотрим произвольный *Ord*-подпуть $\{e''_i\}_{i \in [0..k]}$. Запросы $e''_0$ и $e''_k$ являются запросами записи, т.к. они связаны с другими подпутями $mo$-ребрами. По транзитивности отношения $\mathbf{s}^j.Ord$ (лемма 1) верно $\mathbf{s}^j.Ord(e''_0, e''_k)$, а, значит, подпуть может быть сокращён до этих двух запросов.

Таким образом, кратчайший путь от $e'$ к $e$ в отношении $mo{\restriction_{\mathbf{s}^j.Evt}} \cup \mathbf{s}^j.Ord$ проходит только через запросы записи. Отношение $\mathbf{s}^n.Ord$ содержит все $mo$-ребра по определению $mo$. Также оно содержит все *Ord*-ребра из кратчайшего пути по леммам 3 и 4. Из этого следует, что выбранный цикл присутствует в $\mathbf{s}^n.Ord$, что противоречит ацикличности $\mathbf{s}^n.Ord$. □

### 3.6.3 Определение машины ARM+τ

По сравнению с ARM-машиной состояние машины ARM+τ обладает одной дополнительной компонентой $H : Tid \times Path \rightharpoonup \texttt{Time} \times 2^{ReqSet} \times \mathcal{V}$. Эта компонента является частичной функцией, определяемой на экземплярах инструкций записи, которые завершили своё исполнение. Каждому такому экземпляру $(tid, path)$ функция сопоставляет (*i*) метку времени, (*ii*) множество запросов записи и барьеров $S$ в подсистеме памяти, которые гарантировано предшествуют в смысле отношения *Ord* запросу экземпляра $(tid, path)$, если такой запрос присутствует в подсистеме, и (*iii*) фронт сообщения в стиле обещающей модели: по локации $\ell$ он возвращает метку времени, которая является максимальной среди запросов записи в эту локацию из множества $S$. Для краткости мы определяем

$$\frac{Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{label} \mathbf{s}' \qquad label \notin \{\textbf{Propagate}\ \_\ \_, \textbf{Write commit}\ \_\ \_\ \_\ \_\}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[ARM{+}\tau]{label} \langle \mathbf{s}', H\rangle}$$

$$\frac{Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Propagate}\ e\ tid} \mathbf{s}' \qquad \mathbf{s}'.Ord \cup \mathsf{tedges}(\mathbf{s}'.Evt, H_\tau)\ \text{ациклично}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[ARM{+}\tau]{\textbf{Propagate}\ e\ tid} \langle \mathbf{s}', H\rangle}$$

$$\frac{\begin{array}{c} Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Write commit}\ tid\ path\ \ell\ val} \langle M'_{\text{POP}}, iordf', tapef'\rangle \\ tapef'(tid, path) = \mathsf{W}\ (\mathsf{com}\ im\ \ell\ val) \qquad tape \triangleq \mathbf{s}.tapef(tid) \\ \mathsf{time\text{-}range}(im, \ell, \tau, tid, path, tape, H) \qquad \mathsf{coherent\text{-}thread}(\ell, \tau, path, tape, H) \\ \mathsf{uniq\text{-}time\text{-}loc}(\ell, \tau, \mathbf{s}.tapef, H) \qquad path^{\mathtt{sy}} \triangleq \mathsf{last}_{\mathtt{sy}}(tape, path) \\ S \triangleq \underline{\text{if}}\ path^{\mathtt{sy}} \neq [] \\ \quad \underline{\text{then}}\ \{\langle tid, path^{\mathtt{sy}}, \mathsf{dmb}\rangle\} \cup \mathsf{prev\text{-}Ord\text{-}req}(tid, path^{\mathtt{sy}}, tape, H) \\ \quad \underline{\text{else}}\ \varnothing \\ S' \triangleq \underline{\text{if}}\ im\ \underline{\text{then}}\ S \cup \{\langle tid, path, \mathsf{wr}\ \ell : val\rangle\}\ \underline{\text{else}}\ S \\ view \triangleq [\ell@\tau] \sqcup \mathbf{viewf}(tid, path^{\mathtt{sy}}, path^{\mathtt{sy}}, tape, H) \\ H' \triangleq H[(tid, path) \mapsto (\tau, S', view)] \\ \quad M'_{\text{POP}}.Ord \cup \mathsf{tedges}(M'_{\text{POP}}.Evt, H'_\tau)\ \text{ациклично} \end{array}}{Prog \vdash \langle \mathbf{s}, H\rangle \xrightarrow[ARM{+}\tau]{\textbf{Write commit}\ tid\ path\ \ell\ val\ \tau} \langle\langle M'_{\text{POP}}, iordf', tapef'\rangle, H'\rangle}$$

Рис. 28. Переходы машины ARM+τ

проекции компоненты $H$: $H_\tau : Tid \times Path \rightharpoonup \mathtt{Time}$, $H_{\leqslant} : Tid \times Path \rightharpoonup 2^{ReqSet}$, и $H_{\mathsf{view}} : Tid \times Path \rightharpoonup \mathcal{V}$. Компонента $H$ начального состояния машины $ARM + \tau$, $\mathbf{a}^{\text{init}} \triangleq \langle \mathbf{s}^{\text{init}}, H^{\text{init}}\rangle$, присваивает нулевые метки времени всем инициализирующим записям: $H^{\text{init}} \triangleq [(0_{tid}, []) \mapsto \langle \mathbf{0}, \lambda\ell.\ \mathbf{0}, \varnothing\rangle]$.

Правила переходов машины ARM+τ приведены на рис. 28. Первое правило транслирует все переходы ARM-машины в переходы машины ARM+τ, кроме **Уведомление потока** и **Завершение записи**; при этом компонента $H$ в правиле не используется. Новое правило **Завершение записи** обновляет компоненту $H$, и оба правила **Уведомление потока** и **Завершение записи** имеют общее ограничение — объединение отношения *Ord* и отношения tedges, построенного по меткам

времени, должно быть ациклично. При этом tedges определяется так:

$$\mathsf{tedges}(Evt, H_\tau) \triangleq$$
$$\{(e, e') \in Evt \times Evt \mid$$
$$e, e' \text{ — запросы записи}, e.loc = e'.loc, H_\tau(e.tid, e.path) < H_\tau(e'.tid, e'.path)\}.$$

Остальные правила не могут добавить циклов в упомянутое объединение, поэтому в них не требуется такая проверка.

Рассмотрим подробнее правило **Завершение записи**. Оно выбирает метку времени $\tau$, которая должна быть уникальна среди запросов записи в локацию $\ell$ (предикат uniq-time-loc). Кроме того, эта метка должна быть согласована с метками времени выполненных экземпляров записи в эту локацию (предикат coherent-thread): $\tau$ должно быть больше, чем метки времени предшествующих экземпляров, и меньше последующих. Правило **Завершение записи** ARM-машины не отправляет запрос на запись ($im = \mathit{false}$), если существует следующий завершенный экземпляр записи в ту же локацию. Тем не менее, машина $ARM + \tau$ присваивает метку времени завершаемому экземпляру записи и в таком случае. При этом метка времени экземпляра, запрос которого был отправлен в подсистему памяти, имеет целочисленное значение, а метка времени экземпляра без запросов в подсистеме находится в интервале $(\tau' - 1, \tau')$, где $\tau'$ — это метка времени ближайшего последующего завершенного экземпляра записи в ту же локацию, чей запрос был отправлен (предикат time-range).

Если запрос на запись, который мы обозначим $w$, отправлен в подсистему хранения ($im = \mathit{true}$), и существует предшествующий экземпляр $\mathtt{sy}$-барьера ($\mathit{path}^{\mathtt{sy}} \neq []$), то множество запросов, которые гарантировано предшествуют $w$ в отношении *Ord* (функция $H_\leqslant$), включает три набора запросов. Первый набор состоит из запроса предшествующего барьера $\langle \mathit{tid}, \mathit{path}^{\mathtt{sy}}, \mathsf{dmb}\rangle$. Второй набор включает запросы экземпляров записи, предшествующие экземпляру $\mathit{path}^{\mathtt{sy}}$, prev-Ord-req$(\mathit{tid}, \mathit{path}^{\mathtt{sy}}, \mathit{tape}, H)$. Третий набор является объединением множеств $H_\leqslant(e.\mathit{tid}, e.\mathit{path})$ за каждый запрос записи $e$, прочитанный потоком до экземпляра $\mathit{path}^{\mathtt{sy}}$:

$$\mathsf{prev\text{-}Ord\text{-}req}(\mathit{tid}, \mathit{path}^{\mathtt{sy}}, \mathit{tape}, H) \triangleq$$
$$\{\langle \mathit{tid} : \mathit{path}'@\ell', \mathit{val}'\rangle \mid \mathit{path}' < \mathit{path}^{\mathtt{sy}}, \mathit{tape}(\mathit{path}') = \mathsf{W}\ (\mathsf{com}\ \_\ \ell'\ \mathit{val}')\} \cup$$
$$\bigcup\{H_\leqslant(e.\mathit{tid}, e.\mathit{path}) \mid \mathit{path}'' < \mathit{path}^{\mathtt{sy}}, \mathit{tape}(\mathit{path}'') = \mathsf{R}\ (\mathsf{sat}\ \mathsf{com}\ e)\}.$$

Почему все эти запросы гарантировано предшествуют запросу $w$ в отношении *Ord*? Во-первых, все запросы, кроме $f \triangleq \langle \mathit{tid}, \mathit{path}^{\mathtt{sy}}, \mathsf{dmb}\rangle$, предшествуют

$f$, т.к. в момент добавления $f$ подсистема памяти добавила по *Ord*-ребру $(e, f)$ за каждый запрос $e$, о котором был осведомлён поток. Последнее верно в силу следующих рассуждений. О каждом запросе записи $e'$, который был отправлен потоком до отправки $f$, поток естественным образом осведомлён. О каждом запросе записи $e''$, который был прочитан потоком до отправки $f$, поток осведомлён по ограничениям подсистемы памяти. Эти ограничения гарантируют, что запрос чтения может получить ответ из запроса записи, только если о них осведомлены одни и те же потоки.

Во-вторых, когда поток отправляет запрос $w$, он добавляет ребро $(f, w)$ в отношение *Ord*. Значит все упомянутые выше запросы также предшествуют $w$ в отношении *Ord* в силу транзитивности *Ord*.

Новое вхождение в функцию $H_{\mathsf{view}}$ определяется как поточечный максимум (операция $\sqcup$) на фронтах $[\ell@\tau]$ и $\mathbf{viewf}(tid, path^{\mathtt{sy}}, path^{\mathtt{sy}}, tape, H)$, где

$$\mathbf{viewf}(tid, path^{\mathsf{write}}, path^{\mathsf{read}}, tape, H) \triangleq \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid, path^{\mathsf{write}}, tape, H) \sqcup \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(path^{\mathsf{read}}, tape, H)$$

определяет фронт запросов из вхождения $H_{\leqslant}$:

$$\mathsf{com\text{-}writes\text{-}time}(tid, path, tape, H) \triangleq \{[\ell@\tau] \mid path' < path, tape(path') = \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ \_), \tau = H_\tau(tid, path')\} \cup \{[\ell@\tau] \mid tid', path', path'' < path, \tau = H_\tau(tid', path') \neq \bot, tape(path'') = \mathsf{R}\ (\mathsf{sat}\ sat\text{-}state\ \langle tid', path', \mathsf{wr}\ \ell : \_\rangle), sat\text{-}state \neq \mathsf{inflight}\}.$$

$$\mathsf{sat\text{-}reads\text{-}view}(path, tape, H) \triangleq \{H_{\mathsf{view}}(tid', path') \neq \bot \mid \exists \ell, tid', path', path'' < path, tape(path'') = \mathsf{R}\ (\mathsf{sat}\ sat\text{-}state\ \langle tid', path', \mathsf{wr}\ \ell : \_\rangle), sat\text{-}state \neq \mathsf{inflight}\}.$$

### 3.6.4 Симуляция модели ARMv8 POP

Как было описано в предыдущем разделе, переходы машины ARM+τ являются переходами ARM-машины с дополнительными ограничениями, что потенциально может означать уменьшение допускаемых сценариев поведения. Поскольку это бы означало, что мы не смогли бы использовать ARM+τ как промежуточную машину в доказательстве корректности компиляции, то нужно показать, что все сценарии поведения ARMv8 POP также являются сценариями поведения ARM+τ с точностью до компоненты состояния $H$.

**Теорема 3.** Пусть набор состояний $\{\mathbf{s}^i\}_{i\in[0..n]}$ является исполнением программы *Prog* в ARM-машине. Тогда существует такой набор $\{H^i\}_{i\in[0..n]}$, что $\{\langle \mathbf{s}^i, H^i\rangle\}_{i\in[0..n]}$ является исполнением программы *Prog* в ARM+τ-машине.

$$\begin{aligned}\forall \mathit{Prog}, \{\mathbf{s}^i\}_{i\in[0..n]}.\ \mathbf{s}^0 = \mathbf{s}^{\text{init}} \wedge \mathsf{Final}^{\text{ARM}}(\mathbf{s}^n, \mathit{Prog}) \wedge \\ \mathit{Prog} \vdash \mathbf{s}^0 \xrightarrow[\text{ARM}]{} \ldots \xrightarrow[\text{ARM}]{} \mathbf{s}^n \Rightarrow \exists \{H^i\}_{i\in[0..n]}.\ H^0 = \mathbf{a}^{\text{init}}.H \wedge \\ \mathit{Prog} \vdash \langle \mathbf{s}^0, H^0\rangle \xrightarrow[ARM+\tau]{} \ldots \xrightarrow[ARM+\tau]{} \langle \mathbf{s}^n, H^n\rangle.\end{aligned}$$

*Доказательство.* В разделе 3.6.2 было показано, как по финальному состоянию в сценарии поведения ARMv8 POP построить отношение $mo$ и функцию $\mathsf{map}_\tau : \mathit{req} \rightharpoonup \tau$. Здесь мы также строим их по состоянию $\mathbf{s}^n$, но с небольшим изменением: мы предполагаем, что функция $\mathsf{map}_\tau$ задана на экземплярах инструкций, т.е. на множестве $\mathit{Tid} \times \mathit{Path}$, а не на запросах. Это изменение имеет только стилистический характер, т.к. по запросу однозначно восстанавливается идентификатор экземпляра инструкции, кроме случая инициализирующих запросов.

Мы конструируем множество $\{H^i\}_{i\in[0..n]}$ индуктивно. Начальная компонента $H^0$ равна $\mathbf{a}^{\text{init}}.H$. Также мы используем следующий инвариант для состояний сценария поведения ARM+τ:

$$\begin{aligned}&\mathsf{inv}(\mathbf{s}, H) \triangleq \forall \mathit{tid}, \mathit{path}.\\ &\quad(\mathbf{s}.\mathit{tapef}(\mathit{tid}, \mathit{path}) = \mathsf{W}\ (\mathsf{com}\ \mathit{true}\ \_\ \_) \Rightarrow H_\tau(\mathit{tid}, \mathit{path}) = \mathsf{map}_\tau(\mathit{tid}, \mathit{path})) \vee\\ &\quad(\mathbf{s}.\mathit{tapef}(\mathit{tid}, \mathit{path}) \neq \mathsf{W}\ (\mathsf{com}\ \_\ \_\ \_) \Rightarrow H_\tau(\mathit{tid}, \mathit{path}) = \bot).\end{aligned}$$

Инвариант утверждает, что метки времени, полученные в ходе исполнения в ARM+τ, соответствуют функции $\mathsf{map}_\tau$. Предположим, что мы сконструировали $i$ первых состояний сценария поведения ARM+τ, и инвариант выполняется на них. Докажем индукционный переход, рассмотрев различные варианты шага $\mathit{Prog} \vdash \mathbf{s}^i \xrightarrow[\text{ARM}]{} \mathbf{s}^{i+1}$.

**Уведомление потока.** Выберем $H^{i+1}$ равным $H^i$. Тогда утверждение $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ выполняется, т.к. выполняется $\mathsf{inv}(\mathbf{s}^i, H^i)$ и $\mathbf{s}^{i+1}.\mathit{tapef} = \mathbf{s}^i.\mathit{tapef}$. В разделе 3.6.2 было доказано, что для любого $j \in [0..n]$ отношение $mo{\restriction}_{\mathbf{s}^j.\mathit{Evt}} \cup \mathbf{s}^j.\mathit{Ord}$ ациклично. Утверждение $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ гарантирует, что функция $mo{\restriction}_{\mathbf{s}^{i+1}.\mathit{Evt}}$ равна $\mathsf{tedges}(\mathbf{s}^{i+1}.\mathit{Evt}, H^{i+1}_\tau)$. Тогда отношение $\mathbf{s}^{i+1}.\mathit{Ord} \cup \mathsf{tedges}(\mathbf{s}^{i+1}.\mathit{Evt}, H^{i+1}_\tau)$ ациклично. Таким образом, все дополнительные ограничения правила **Уведомление потока** машины ARM+τ выполнены, и она может совершить соответствующий переход.

**Завершение записи** *tid path***.** Нам нужно рассмотреть два случая.

Если соответствующий запрос записи отправлен в подсистему памяти, то мы выбираем ему метку времени $\tau$, которая является параметром перехода в машине ARM+$\tau$и равна $\mathsf{map}_\tau(tid, path)$. Мы выбираем $H^{i+1}$ в соответствии с определением этой компоненты в правиле **Завершение записи** машины ARM+$\tau$. Инвариант, очевидно, выполняется по определению. По определению функции $\mathsf{map}_\tau$ метка времени $\tau$ уникальна среди запросов записи в соответствующую локацию. Ацикличность отношения $\mathbf{s}^{i+1}.Ord \cup \mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H^{i+1}_\tau)$ следует из тех же рассуждений, что и в предыдущем случае. Из этого, в свою очередь, следует, что $\tau$ больше всех остальных меток времени запросов записи в ту же локацию среди тех, что были отправлены тем же потоком в подсистему памяти. Метка $\tau$ также больше всех меток времени предшествующих завершенных экземпляров записи в ту же локацию, которые не отправляли запросы, как следствие принципа выбора метки времени. В плёнке потока нет последующих завершенных записей в ту же локацию, т.к. запрос был отправлен, следовательно метка времени $\tau$ корректна относительно других записей потока.

Если запрос записи не был отправлен, то $\mathsf{map}_\tau(tid, path) = \bot$. Известно, что существует последующий экземпляр записи в ту же локацию с меткой времени $\tau'$, который отправил запрос в подсистему памяти. Мы выбираем метку времени $\tau$ из интервала $(\tau' - 1, \tau)$ так, чтобы она была корректна относительно требований перехода машины ARM+$\tau$. Мы выбираем $H^{i+1}$ в соответствии с переходом **Завершение записи** машины ARM+$\tau$. При этом инвариант, очевидно, сохраняется.

**Другие переходы.** Мы выбираем компоненту $H^{i+1}$ равной $H^i$. Поскольку в остальных правилах машины ARM+$\tau$ нет дополнительных ограничений, то машина может сделать соответствующий переход, и при этом инвариант сохранится. □

### 3.6.5 Фронты машины ARM+$\tau$

В разделе 3.2.1 была рассмотрена программа MP-SY-LD. В ней использование барьеров памяти позволяет запретить слабый сценарий поведения. Для таких целей в обещающей модели используются фронты. Для того, чтобы показать,

что обещающая машина может симулировать машину ARM+τ, вводится аналог фронтов для ARM+τ.

Введём фронт $\mathsf{view}_{\mathrm{ARM}}$ — аналог базового фронта обещающей машины:

$$\begin{aligned}&\mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, \mathit{tid}, \mathit{path}) \triangleq \\ &\quad \bigsqcup \mathsf{com\text{-}writes\text{-}time}(\mathit{tid}, \mathit{path}, \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathbf{a}.H_\tau) \sqcup \\ &\quad \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(\mathsf{lastCF}(\mathit{tape}, \mathit{path}), \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathbf{a}.H_{\mathsf{view}})\,.\end{aligned}$$

В отличие от $\mathit{view}_{\mathrm{cur}}$, который определяется в целом для потока, фронт $\mathsf{view}_{\mathrm{ARM}}$ дополнительно параметризован путём *path* из тех же соображений, что и состояние переменных в ARMv8 POP параметризовано путём: машина ARM+τ может исполнять экземпляры не по порядку, поэтому разные экземпляры могут иметь разные ограничения, связанные с метками времени. Определение фронта $\mathsf{view}_{\mathrm{ARM}}$ похоже на определение компоненты $H_{\mathsf{view}}$ в правиле **Завершение записи**: этот фронт является композицией одинарных фронтов $[\ell@\tau]$, где $\ell$ и $\tau$ — параметры завершенного экземпляра записи, который предшествует *path*, со значениями $H_{\mathsf{view}}$, связанными с запросами записи, которые прочитаны потоком до последнего выполненного барьера $\mathsf{lastCF}(\mathit{tape}, \mathit{path})$, предшествующего *path*. При этом учитываются и `ld`, и `sy` барьеры, так как они оба могут быть использованы как результат компиляции приобретающего барьера в обещающей модели.

С помощью определения $\mathsf{view}_{\mathrm{ARM}}$ нужные ограничения задаются следующим образом. Если экземпляр чтения $(\mathit{tid}, \mathit{path})$ получил ответ от завершенного экземпляра записи, то метка времени этой записи не меньше соответствующего значения $\mathsf{view}_{\mathrm{ARM}}$ для пути *path*. Мы не ограничиваем чтения, которые получили ответ от ещё не завершенной записи, так как такие записи не обладают метками времени. Аналогично, каждый завершенный экземпляр записи имеет метку времени, превосходящую соответствующее значение $\mathsf{view}_{\mathrm{ARM}}$. Упомянутые ограничения формулируются в виде следующей теоремы.

**Теорема 4.** Пусть состояние ARM+τ-машины **a** достижимо из начального. Тогда метка времени завершённого экземпляра записи в локацию $\ell$ не меньше, чем значение фронта $\mathsf{view}_{\mathrm{ARM}}$ по этой локации для экземпляра чтения, прочитавшего из этой записи. Кроме того, метка времени завершённого экземпляра записи больше,

чем значение фронта $\mathsf{view}_{\mathrm{ARM}}$ для самого экземпляра записи.

$$\forall \mathit{Prog}, \mathbf{a}, \mathit{tid}, \mathit{tape} = \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathit{path}.\ \mathit{Prog} \vdash \mathbf{a}^{\mathrm{init}} \xrightarrow[ARM+\tau]{}{}^{*}\ \mathbf{a} \Rightarrow$$
$$(\forall e.\ \mathit{tape}(\mathit{path}) = \mathsf{R}\ (\mathsf{sat}\ \_\ e) \wedge \mathbf{a}.\mathit{tapef}(e.\mathit{tid}, e.\mathit{path}) \text{ завершён} \Rightarrow$$
$$\mathbf{a}.H_{\tau}(e.\mathit{tid}, e.\mathit{path}) \geqslant \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, \mathit{tid}, \mathit{path}, e.\ell)) \wedge$$
$$(\forall \ell.\ \mathit{tape}(\mathit{path}) = \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ \_) \Rightarrow \mathbf{a}.H_{\tau}(\mathit{tid}, \mathit{path}) > \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, \mathit{tid}, \mathit{path}, \ell)).$$

*Доказательство.* Утверждение доказывается с помощью индукции по сценарию поведения $\mathit{Prog} \vdash \mathbf{a}^{\mathrm{init}} \xrightarrow[ARM+\tau]{}{}^{*}\ \mathbf{a}$. □

## 3.7 Доказательство корректности компиляции

В этом разделе доказывается основная теорема главы, представленная в разделе 3.3.

**Теорема 1.** Для любой программы *Prog* и её сценария поведения в модели ARMv8 POP, $\mathit{Prog} \vdash \mathbf{s}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}{}^{*}\ \mathbf{s}$, где $\mathbf{s}$ — финальное состояние, $\mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}, \mathit{Prog})$, существует сценарий поведения этой программы в обещающей модели, $\mathit{Prog} \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^{*}\ \mathbf{p}$, где $\mathbf{p}$ — финальное состояние $\mathsf{Final}^{\mathsf{Promise}}(\mathbf{p}, \mathit{Prog})$; при этом состояния памяти в $\mathbf{s}$ и $\mathbf{p}$ совпадают, $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

*Доказательство.* Зафиксируем программу *Prog*. Используя теорему 3, мы меняем утверждение, которое нужно доказать, на симуляцию обещающей машиной машины ARM+τ:

$$\forall \mathbf{a}.\ \mathit{Prog} \vdash \mathbf{a}^{\mathrm{init}} \xrightarrow[ARM+\tau]{}{}^{*}\ \mathbf{a} \wedge \mathsf{Final}^{ARM+\tau}(\mathbf{a}, \mathit{Prog}) \Rightarrow$$
$$\exists \mathbf{p}.\ \mathit{Prog} \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathsf{Promise}]{}{}^{*}\ \mathbf{p} \wedge \mathsf{Final}^{\mathsf{Promise}}(\mathbf{p}, \mathit{Prog}) \wedge \mathsf{same\text{-}memory}(\mathbf{a}, \mathbf{p}).$$

Для того, чтобы доказать это утверждение, введём набор отношений, связывающих состояния машин. Эти отношения в дальнейшем войдут в отношение симуляции.

Следуя представлению обещающей машины в [23], добавим в состояние каждого потока обещающей машины указатель на следующий к исполнению экземпляр инструкции, который будем обозначать $\mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path}$, и функцию состояния локальных переменных $\mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{st}$. В представлении из раздела 3.5 эти части состояния скрыты помеченной системой переходов.

Отношение $\mathcal{I}_{\mathrm{prefix}}$ устанавливает, что все экземпляры инструкций, выполненные обещающей машиной, завершены машиной ARM+τ:

$$\mathcal{I}_{\mathrm{prefix}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \mathit{tid}, \mathit{path}' < \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path}.\ (\mathbf{a}.\mathit{tapef}(\mathit{tid}, \mathit{path}') \text{ завершено}).$$

Следующие отношения связывают память обещающей машины с подсистемой памяти ARM+τ. Отношение $\mathcal{I}_{\textbf{mem1}}$ утверждает, что для каждого завершенного экземпляра записи в ARM+τ существует сообщение в памяти обещающей машины с теми же локацией, значением и меткой времени, кроме того фронт сообщения не больше, чем соответствующий фронт запроса в ARM+τ. При этом, если путь экземпляра записи меньше, чем указатель обещающей машины, то сообщение было обещано, и обещание выполнено, иначе — только обещано:

$$\begin{aligned}
&\mathcal{I}_{\textbf{mem1}}(\mathbf{a},\mathbf{p}) \triangleq \forall tid, \ell, val, \tau, view', path.\\
&\quad \mathsf{W}\ (\mathsf{com}\ _\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \langle \tau, _, view' \rangle = \mathbf{a}.H(tid, path) \Rightarrow\\
&\quad \exists view \leqslant view'.\\
&\qquad (path \geqslant \mathbf{p}.\mathcal{TS}(tid).path \Rightarrow \langle \ell : val@\tau, view \rangle \in \mathbf{p}.\mathcal{TS}(tid).promises) \wedge\\
&\qquad (path < \mathbf{p}.\mathcal{TS}(tid).path \Rightarrow\\
&\qquad\quad \langle \ell : val@\tau, view \rangle \in \mathbf{p}.M \setminus \bigcup_{tid} \mathbf{p}.\mathcal{TS}(tid).promises).
\end{aligned}$$

Отношение $\mathcal{I}_{\textbf{mem2}}$ связывает памяти машин в обратном направлении: для каждого сообщения из памяти обещающей машины в плёнке ARM+τ есть соответствующий завершённый экземпляр записи:

$$\begin{aligned}
&\mathcal{I}_{\textbf{mem2}}(\mathbf{a},\mathbf{p}) \triangleq \forall \langle \ell : val@\tau, view \rangle \in \mathbf{p}.M.\ \tau \neq \mathbf{0} \Rightarrow \exists tid, path, view' \geqslant view.\\
&\quad \mathsf{W}\ (\mathsf{com}\ _\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \mathbf{a}.H(tid, path) = \langle \tau, _, view' \rangle.
\end{aligned}$$

Отношение $\mathcal{I}_{\textbf{mem3}}$ связывает инициализирующие записи:

$$\mathcal{I}_{\textbf{mem3}}(\mathbf{a},\mathbf{p}) \triangleq \forall \ell.\ \langle 0_{tid}, [], \mathsf{wr}\ \ell : 0 \rangle \in \mathbf{a}.M_{\mathrm{POP}} \wedge \langle \ell : 0@\mathbf{0}, \lambda \ell.\ \mathbf{0} \rangle \in \mathbf{p}.M.$$

Отношение $\mathcal{I}_{\textbf{view}}$ утверждает, что фронты обещающей машины ограничены комбинацией фронтов, связанных с завершёнными экземплярами чтений и записей машины ARM+τ. Для приобретающего фронта считаются все экземпляры записи и чтения, путь которых меньше *path*. Для базового фронта считаются все экземпляры записи, путь которых меньше *path*, и чтения, путь которых меньше последнего завершённого экземпляра `ld` барьера ($\mathsf{last}\texttt{ld}(tape, path)$). Для высвобождающего фронта учитываются все экземпляры записи вплоть до последнего завершённого экземпляра `sy` барьера ($\mathsf{last}\texttt{sy}(tape, path)$) и все экземпляры чтения до завершённого экземпляра `ld` барьера, предшествующего этому `sy` барьеру

(last$\texttt{ldsy}(\mathit{tape}, \mathit{path})$).

$$\begin{aligned}
&\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \mathit{tid}, \mathit{tape} = \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathit{path} = \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path}.\\
&\quad \underline{\text{let}}\ \mathit{path}^{\texttt{ld}}, \mathit{path}^{\texttt{sy}} \triangleq \mathsf{last}\texttt{ld}(\mathit{tape}, \mathit{path}), \mathsf{last}\texttt{sy}(\mathit{tape}, \mathit{path})\ \underline{\text{in}}\\
&\quad \underline{\text{let}}\ \mathit{path}^{\texttt{ldsy}} \triangleq \mathsf{last}\texttt{ldsy}(\mathit{tape}, \mathit{path})\ \underline{\text{in}}\\
&\quad (\mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{view}_{\text{acq}} \leqslant \bigsqcup \textbf{viewf}(\mathit{tid}, \mathit{path}, \mathit{path}, \mathit{tape}, \mathbf{a}.H)) \wedge\\
&\quad (\mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{view}_{\text{cur}} \leqslant \bigsqcup \textbf{viewf}(\mathit{tid}, \mathit{path}^{\texttt{ld}}, \mathit{path}, \mathit{tape}, \mathbf{a}.H)) \wedge\\
&\quad (\mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{view}_{\text{rel}} \leqslant \bigsqcup \textbf{viewf}(\mathit{tid}, \mathit{path}^{\texttt{ldsy}}, \mathit{path}^{\texttt{sy}}, \mathit{tape}, \mathbf{a}.H).
\end{aligned}$$

Отношение $\mathcal{I}_{\textbf{state}}$ связывает состояние локальных переменных в потоке обещающей машины со значением функции состояния машины ARM+τ:

$$\begin{aligned}
&\mathcal{I}_{\textbf{state}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \mathit{tid}, \mathit{regf} = \mathsf{regf}_{\mathsf{com}}(\mathit{Prog}(\mathit{tid}), \mathbf{a}.\mathit{tapef}(\mathit{tid}), \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path}).\\
&\quad \forall \mathit{reg}, \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{st}(\mathit{reg}) = \mathit{regf}(\mathit{reg}).
\end{aligned}$$

Отношение $\mathcal{I}_{\text{com-SY}}$ утверждает, что все экземпляры $\texttt{sy}$ барьера, которые предшествуют завершённому экземпляру записи, выполнены обещающей машиной:

$$\begin{aligned}
&\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \mathit{tid}, \mathit{tape} = \mathbf{a}.\mathit{tapef}(\mathit{tid}),\\
&\quad \mathit{path}_{\mathit{write}} = \mathsf{last\text{-}write\text{-}com}(\mathit{tape}), \mathit{path}_{\texttt{sy}} < \mathit{path}_{\mathit{write}}.\\
&\quad\quad \mathit{tape}(\mathit{path}_{\texttt{sy}}) = \mathsf{F}\ _\ \texttt{sy} \Rightarrow \mathit{path}_{\texttt{sy}} < \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path},
\end{aligned}$$

где $\mathsf{last\text{-}write\text{-}com}(\mathit{tape})$ — это путь последнего завершённого потоком экземпляра записи. Данное отношение нужно для доказательства сертифицируемости шагов обещающей машины, участвующих в симуляции.

Отношение $\mathcal{I}_{\text{reach}}$ показывает, что состояния машин достижимы из начального:

$$\mathcal{I}_{\text{reach}}(\mathbf{a}, \mathbf{p}) \triangleq \mathit{Prog} \vdash \mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*}\ \mathbf{a} \wedge \mathit{Prog} \vdash \mathbf{p}^{\text{init}} \xrightarrow[\text{Promise}]{}{}^{*}\ \mathbf{p}.$$

Отношение $\mathcal{I}_{\text{base}}$ комбинирует вышеупомянутые отношения:

$$\mathcal{I}_{\text{base}} \triangleq \mathcal{I}_{\text{prefix}} \cap \mathcal{I}_{\textbf{mem1}} \cap \mathcal{I}_{\textbf{mem2}} \cap \mathcal{I}_{\textbf{mem3}} \cap \mathcal{I}_{\textbf{view}} \cap \mathcal{I}_{\textbf{state}} \cap \mathcal{I}_{\text{com-SY}} \cap \mathcal{I}_{\text{reach}}.$$

При симуляции в каждый момент верно одно из двух: либо обещающая машина ожидает, что машина ARM+τ завершит исполнение экземпляра инструкции, либо указатель одного (и только одного) из потоков обещающей машины указывает на завершённый экземпляр инструкции. Последнее означает, что обещающая машина должна сделать шаг, соответствующий этому экземпляру. Для того,

чтобы задать оба возможных варианта, мы используем вспомогательный предикат:

$$\mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid,\mathbf{a},\mathbf{p}) \triangleq \underline{\text{let}}\ tape, path \triangleq \mathbf{a}.tapef(tid), \mathbf{p}.\mathcal{TS}(tid).path\ \underline{\text{in}}$$
$$tape(path) = \bot \vee tape(path) \text{ не завершен.}$$

С его помощью случай, когда обещающая машина ожидает, выражается предикатом $\mathcal{I}_{\text{Promise is up to ARM}}$, а случай, когда обещающая машина должна сделать шаг — предикатом $\mathcal{I}_{\text{Promise isn't up to ARM}}$:

$$\mathcal{I}_{\text{Promise is up to ARM}} \quad (\mathbf{a},\mathbf{p}) \triangleq \forall \quad tid. \quad \mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid,\mathbf{a},\mathbf{p}).$$
$$\mathcal{I}_{\text{Promise isn't up to ARM}} \quad (\mathbf{a},\mathbf{p}) \triangleq \exists! \quad tid. \neg \quad \mathcal{I}^{tid}_{\text{Promise is up to ARM}}(tid,\mathbf{a},\mathbf{p}).$$

Данные предикаты используются для определения двух отношений симуляции:

$$\mathcal{I}_{\text{pre}} \triangleq \mathcal{I}_{\text{base}} \cap \mathcal{I}_{\text{Promise isn't up to ARM}} \qquad \mathcal{I} \triangleq \mathcal{I}_{\text{base}} \cap \mathcal{I}_{\text{Promise is up to ARM}}$$

Если состояния машин связаны отношением $\mathcal{I}_{\text{pre}}$, то существует поток обещающей машины, который может сделать переход. После этого перехода состояния машин снова связаны отношением $\mathcal{I}_{\text{pre}}$, и тот же поток может сделать ещё один шаг, или состояния связаны отношением $\mathcal{I}$, и все потоки обещающей машины ждут. Это утверждение сформулировано в лемме 5.

**Лемма 5.** $\forall(\mathbf{a},\mathbf{p}) \in \mathcal{I}_{\text{pre}}.\ \exists\mathbf{p}'.Prog \vdash \mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}' \wedge (\mathbf{a},\mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}.$

Доказательство леммы 5 приведено в приложении В.

Как следствие того, что в каждом корректном состоянии машины ARM+τ существует лишь конечное число завершённых экземпляров инструкций, можно показать, что обещающая машина может за конечное число шагов "нагнать" ARM+τ (лемма 6).

**Лемма 6.** $\forall(\mathbf{a},\mathbf{p}) \in \mathcal{I}_{\text{pre}}.\ \exists\mathbf{p}'.\ Prog \vdash \mathbf{p} \xrightarrow[\text{Promise}]{}{}^{*} \mathbf{p}' \wedge (\mathbf{a},\mathbf{p}') \in \mathcal{I}.$

*Доказательство.* Зафиксируем **a** и **p**. Из того, что $(\mathbf{a},\mathbf{p}) \in \mathcal{I}_{\text{pre}}$, следует $\mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a},\mathbf{p})$. Т.о. известно, что существует единственный поток *tid*, его плёнка $tape = \mathbf{a}.tapef(tid)$ и путь $path = \mathbf{p}.\mathcal{TS}(tid).path$, что исполнение экземпляра $tape(path)$ завершено. Зафиксируем $tid, tape, path, cmds = Prog(tid)$.

Из того, что $tape(path)$ завершён, следует, что существует конечный набор $\{path_i\}_{i\in[0,n]}$ такой, что:

– $\forall i, tape(path_i)$ завершено;

– $\forall i < n, \mathit{path}_{i+1} \in \mathsf{next\text{-}path}(\mathit{path}_i, \mathit{cmds})$;
– $\forall \mathit{path}' \in \mathsf{next\text{-}path}(\mathit{tape}, \mathit{cmds}), \mathit{tape}(\mathit{path}') = \bot \vee \mathit{tape}(\mathit{path}')$ не завершено.

Здесь $\mathsf{next\text{-}path}(\mathit{path}_i, \mathit{cmds})$ является функцией, которая по пути экземпляра инструкции восстанавливает путь до непосредственно следующего за ним экземпляра (см. формальное определение функции в приложении Б).

Далее лемма доказывается индукцией по $n$ с помощью леммы 5. □

Предположим, что состояния обещающей и ARM+τ машин связаны отношением $\mathcal{I}$. Для такой ситуации мы показываем, что машина ARM+τ может сделать шаг. Если этот шаг имеет тип **Завершение записи**, то обещающая машина может пообещать соответствующее сообщение, и новые состояния машин будут связаны отношением $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$. Если это другой шаг, то обещающая машина не делает своего ответного шага, но состояния машин опять же связаны отношением $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$.

**Лемма 7.** $\forall (\mathbf{a}, \mathbf{p}) \in \mathcal{I}$.
$(\forall \mathbf{a}'.\ \mathit{Prog} \vdash \mathbf{a} \xrightarrow[ARM+\tau]{\neg\ \textbf{Write commit}} \mathbf{a}' \Rightarrow (\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}) \wedge$
$(\forall \mathbf{a}'.\ \mathit{Prog} \vdash \mathbf{a} \xrightarrow[ARM+\tau]{\textbf{Write commit}} \mathbf{a}' \Rightarrow \exists \mathbf{p}'.\ \mathit{Prog} \vdash \mathbf{p} \xrightarrow[\mathsf{Promise}]{\textbf{Promise write}} \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I})$.

Доказательство леммы 7 приведено в приложении В.

Основная лемма симуляции непосредственно следует из лемм 5, 6 и 7.

**Лемма 8.** Пусть состояние ARM+τ-машины **a** и обещающей машины **p** связаны отношением $\mathcal{I}$, а состояние $\mathbf{a}'$ достижимо за один шаг из **a**. Тогда существует такое состояние $\mathbf{p}'$, достижимое из **p**, что $\mathbf{a}'$ и $\mathbf{p}'$ также связаны отношением $\mathcal{I}$.

$$\forall (\mathbf{a}, \mathbf{p}) \in \mathcal{I}, \mathbf{a}'.\ \mathbf{a} \xrightarrow[ARM+\tau]{} \mathbf{a}' \Rightarrow \exists \mathbf{p}'.\ \mathit{Prog} \vdash \mathbf{p} \xrightarrow[\mathsf{Promise}]{}{}^{*}\ \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}.$$

Вернёмся к доказательству теоремы 1. Она доказывается индукцией по длине сценария поведения с помощью леммы 8. Часть утверждения теоремы о том, что конечные состояния памяти совпадают, напрямую следует из отношений $\mathcal{I}_{\textbf{mem1}}$ и $\mathcal{I}_{\textbf{mem2}}$. Единственное, что осталось доказать, что состояние **p** является финальным состоянием обещающей машины ($\mathsf{Final}^{\mathsf{Promise}}(\mathbf{p}, \mathit{Prog})$).

Обещающая машина не может сделать нового перехода, т.к. это бы означало, что машина ARM+τ может загрузить новый экземпляр инструкции, а это противоречило бы $\mathsf{Final}^{ARM+\tau}(\mathbf{a}, \mathit{Prog})$. Кроме того, все сообщения из состояния **p** были обещаны и выполнены, потому что выполняется $\mathcal{I}_{\textbf{mem1}}(\mathbf{a},\mathbf{p})$ и $\mathcal{I}_{\textbf{mem2}}(\mathbf{a}, \mathbf{p})$. □

## 3.8 Выводы

В главе представлено доказательство корректности компиляции из обещающей модели памяти в модель памяти ARMv8 POP. Это доказательство является важным аргументом в пользу использования обещающей модели как замены существующих моделей памяти промышленных языков программирования. Само доказательство представляет научный интерес, т.к. использованные ранее схемы доказательства для случая моделей x86 и Power не применимы к ARMv8 POP.

В доказательстве рассмотрено подмножество обещающей модели памяти, в которое входят расслабленные обращения к памяти и высвобождающие и приобретающие барьеры. Тем не менее, доказательство может быть расширено и для других конструкций обещающей модели.

# Глава 4. Корректность компиляции из обещающей модели в аксиоматическую модель ARMv8.3

В этой главе описывается доказательство корректности эффективной схемы компиляции из подмножества обещающей модели памяти [11] в аксиоматическую модель памяти ARMv8.3 POP [10]. Рассмотренное подмножество обещающей модели состоит из расслабленных обращений к памяти, высвобождающего и приобретающего барьеров памяти без сертификации.

Как было описано в разделе 3.1, доказательство корректности компиляции из обещающей модели памяти является важной задачей. В главе 3 было рассмотрено доказательство для модели ARMv8 POP [9], однако задача для архитектуры ARM оказалась не полностью закрыта — в апреле 2017 года была предложена новая модель памяти ARMv8.3 [10, 109] и потребовалось новое доказательство. Также как и в случае с моделью ARMv8 POP, схема доказательства корректности компиляции, использованная в [11] для моделей x86 и Power, не подходит для модели ARMv8.3. Дело в том, что для использования этой схемы нужно, чтобы модель была представлена в виде оптимизаций над более строгой моделью, и все результаты сценариев поведения в этой более строгой модели должны быть получены также и в обещающей модели, при этом механизм обещаний не должен быть использован. Данное утверждение для модели ARMv8.3 не является истинным. Чтобы убедиться в этом, рассмотрим такую программу:

$$\begin{array}{l|l} a := [x]; \ //1 & [y] := 1; \\ \textbf{if } a = 1 \textbf{ then} & \mathsf{fence}(\mathtt{sy}); \\ \quad b := [y] \ //0 & [x] := 1 \end{array}$$

Между инструкциями чтения в левом потоке имеется зависимость по управлению, а в правом потоке находится барьер памяти, запрещающий неупорядоченное исполнение записей. В рамках модели памяти ARMv8.3 возможен сценарий программы с результатом $a = 1$ и $b = 0$. Следует отметить, что в обоих потоках инструкции не могут быть переставлены. Модель ARMv8.3 является аксиоматической, и это существенно отличает ее от модели ARMv8 POP [9]. Следовательно, невозможно простым способом задать симуляцию модели ARMv8.3 обещающей моделью. Преодолевая это ограничение, автор диссертационного исследования разработал специальный способ обхода графа модели ARMv8.3 (раздел 4.3),

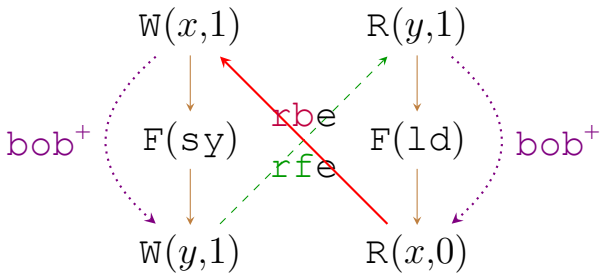



Рис. 29. Сценарий поведения программы MP-sy-ld, запрещённый в модели ARMv8.3

представленный в виде операционной семантики. Этот обход используется при доказательстве симуляции (раздел 4.5).

## 4.1 Модель ARMv8.3

Модель ARMv8.3 [10] является аксиоматической и представляет семантику программы в виде набора графов, где каждый граф соответствует определённому запуску программы. Как и в модели C/C++11 [4] она использует отношения $\mathtt{po}$, "читает из" $\mathtt{rf}$, порядка памяти $\mathtt{mo}$. Кроме того, используется производное отношение "читает ранее" $\mathtt{rb}$ (reads before):

$$\mathtt{rb} \triangleq \mathtt{rf}^{-1}; \mathtt{mo},$$

связывающее событие чтения $a$ с событием записи $b$, которое $\mathtt{mo}$-следует за записью, из которой читает $a$. Композиция отношений ';' задаётся следующим образом:

$$A; B \triangleq \{\langle a, b\rangle \mid \exists c.\ \langle a, c\rangle \in A \wedge \langle c, b\rangle \in B\}.$$

Добавление в программу MP барьеров памяти, как и в случае с моделью ARMv8 POP, делает невозможным результат $[a = 1, b = 0]$ (см. рис. 29):

$$\begin{array}{l|l} [x] := 1; & a := [y]; \ //1 \\ \mathsf{fence(sy)}; & \mathsf{fence(ld)}; \\ [y] := 1; & b := [x]; \ //0 \end{array} \qquad \text{(MP-sy-ld)}$$

В данном графе опущены инициализирующие записи и добавлено отношение барьеров $\mathtt{bob}$ (barrier-ordered-before), которое задаётся так:

$$\mathtt{bob} \triangleq \mathtt{po}; [\mathtt{F(sy)}] \cup [\mathtt{F(sy)}]; \mathtt{po} \cup [\mathtt{R}]; \mathtt{po}; [\mathtt{F(ld)}] \cup [\mathtt{F(ld)}]; \mathtt{po},$$

где $[A] \triangleq \{(a,a) \mid a \in A\}$. Также в графе заменены ребра отношений rf и rb на rfe и rbe, где e (external) означает, что отношение связывает только события, принадлежащие разным потокам. В представленном графе имеется цикл: bob, rfe, rbe, и это противоречит одной из аксиом, используемой в модели ARMv8.3 для задания ARM-согласованного сценария поведения. Следовательно, данный граф не является ARM-согласованным, и результат $[a = 1, b = 0]$ запрещен для программы MP-sy-ld. Ниже представлено формальное определение сценария поведения в модели ARMv8.3 [10].

**Определение 5.** *Сценарий поведения* $G$ является графом, включающий следующие компоненты.

1. Конечное множество *событий* $\mathtt{E} \subseteq \mathbb{N}$, которое включает начальные события $\mathtt{E}_0 = \{a_0^x \mid x \in Loc\}$. Для обозначения событий используются символы $a$,$b$, ...
2. Функция tid, возвращающая по событию из E номер потока, создавшего это событие; любое начальное событие считается относящимся к потоку с номером 0, $\mathtt{tid}(a_0^x) = 0$.
3. Функция lab, присваивающая каждому событию из E *метку*; метки могут обозначать следующие операции:
   - *чтение*, $\mathtt{R}(x,v)$, где $x \in Loc$ — это *локация*, из которой событие читает *значение* $v \in Val$;
   - *запись*, $\mathtt{W}(x,v)$, где $x \in Loc$ — это *локация*, в которую событие записывает *значение* $v \in Val$;
   - *барьер памяти*, $\mathtt{F}(o)$, где $o \in \{\mathtt{ld},\mathtt{sy}\}$ — это *тип барьера*, причём sy-барьеры более строгие, чем ld, $\mathtt{ld} \sqsubset \mathtt{sy}$.

   Каждое начальное событие является инициализирующей записью в определённую локацию:

   $$\forall a_0^x \in \mathtt{E}_0.\ \mathtt{lab}(a_0^x) = \mathtt{W}(x,0).$$

   Функция lab определяет следующие частичные функции, которые по событию возвращают:
   - typ — тип события (R,W или F);
   - mod — тип барьера;
   - loc — целевую локацию;
   - $\mathtt{val}_\mathtt{r}$ — прочитанное значение;

– $\mathtt{val}_{\mathtt{w}}$ — записанное значение.

Далее мы будем использовать символы $\mathtt{R}, \mathtt{W}$ и $\mathtt{F}$ в том числе и для обозначения соответствующих множеств событий, например, $\mathtt{R}$ — для обозначения множества $\{e \in \mathtt{E} \mid \mathtt{typ}(e) = \mathtt{R}\}$.

4. Строгий частичный порядок на событиях $\mathtt{po} \subseteq \mathtt{E} \times \mathtt{E}$, называемый *программным порядком*; он является объединением непересекающихся множеств $\{\mathtt{po}_i\}_{i \in \{0\} \cup Tid}$, где $\mathtt{po}_0 = \mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0)$, и для каждого потока $i \in Tid$ отношение $\mathtt{po}_i$ является полным порядком на множестве его событий $\{a \in \mathtt{E} \mid \mathtt{tid}(a) = i\}$.
5. Отношения $\mathtt{data}, \mathtt{ctrl}, \mathtt{addr} \subseteq \mathtt{po} \backslash \mathtt{po}_0$, удовлетворяющие следующим ограничениям:
   – $\mathtt{data} \subseteq \mathtt{R} \times \mathtt{W}$;
   – $\mathtt{addr} \subseteq \mathtt{R} \times (\mathtt{R} \cup \mathtt{W})$;
   – $\mathtt{ctrl}; \mathtt{po} \subseteq \mathtt{ctrl}$.

   Они позволяют задавать зависимости по данным, по потоку управления и по целевому адресу инструкции соответственно.
6. Отношение $\mathtt{rf} \subseteq [\mathtt{W}]; =_{\mathtt{loc}}; [\mathtt{R}]$, удовлетворяющее следующим ограничениям: $(i)\ \mathtt{val}_{\mathtt{w}}(a) = \mathtt{val}_{\mathtt{r}}(b)$ для любой пары $\langle a,b \rangle \in \mathtt{rf}$; $(ii)\ a_1 = a_2$, если $\langle a_1,b \rangle, \langle a_2,b \rangle \in \mathtt{rf}$; также вводится функция $\mathtt{rf}^{-1}$, которая определена на области отображения $\mathtt{rf}$, $codom(\mathtt{rf})$, и для любого $a$ из $codom(\mathtt{rf})$ верно следующее: $\langle \mathtt{rf}^{-1}(a), a \rangle \in \mathtt{rf}$.
7. Строгий частичный порядок $\mathtt{mo}$ на элементах $\mathtt{W}$, являющийся объединением отношений $\{\mathtt{mo}_x\}_{x \in Loc}$, где $\mathtt{mo}_x$ — это полный порядок на элементах $\mathtt{W}_x = \{e \in \mathtt{W} \mid \mathtt{loc}(e) = x\}$.

С помощью суффиксов $\mathtt{i}$ и $\mathtt{e}$ обозначаются подмножества отношений, которые связывают события одного и разных потоков соответственно. Например, $\mathtt{moi} = \{\langle w, w' \rangle \in \mathtt{mo} \mid \mathtt{tid}(w) = \mathtt{tid}(w')\}$ и $\mathtt{rfe} = \{\langle w, r \rangle \in \mathtt{rf} \mid \mathtt{tid}(w) \neq \mathtt{tid}(r)\}$.

**Определение 6.** Сценарий поведения $G$ является ARM-согласованным, если выполняются следующие условия:

– $\mathtt{R} = codom(\mathtt{rf})$; (COMPLETENESS)
– отношение $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rb} \cup \mathtt{mo} \cup \mathtt{rf}$ не имеет циклов; (INTERNAL)
– отношение $\mathtt{obs} \cup \mathtt{dob} \cup \mathtt{bob}$ не имеет циклов. (EXTERNAL)

В определении используются следующие производные отношения:

$$\texttt{obs} \triangleq \texttt{rfe} \cup \texttt{rbe} \cup \texttt{moe} \qquad \textit{(observed-by)}$$

$$\texttt{dob} \triangleq \begin{array}{l}(\texttt{addr} \cup \texttt{data}); \texttt{rfi}^? \cup \\ (\texttt{ctrl} \cup \texttt{data}); [\texttt{W}]; \texttt{moi}^? \cup \\ \texttt{addr}; \texttt{po}; [\texttt{W}]\end{array} \qquad \textit{(dependency-ordered-before)}$$

$$\texttt{bob} \triangleq \begin{array}{l}\texttt{po}; [\texttt{F(sy)}] \cup [\texttt{F(sy)}]; \texttt{po} \cup \\ [\texttt{R}]; \texttt{po}; [\texttt{F(ld)}] \cup [\texttt{F(ld)}]; \texttt{po}\end{array} \qquad \textit{(barrier-ordered-before)}$$

## 4.2 Структура доказательства корректности компиляции

Центральным результатом данной главы является следующая теорема.

**Теорема 5.** Для любой программы $Prog$, результата её компиляции $Prog_{\text{ARM}}$ и ARM-согласованного сценария поведения $G$ программы $Prog_{\text{ARM}}$ существует такой сценарий поведения обещающей машины, что финальное состояние памяти машины совпадает с состоянием памяти $G$.

Здесь под “финальным состоянием памяти”, также как и в случае корректности компиляции для ARMv8 POP, понимается, в случае обещающей семантики, значение последних записей в локации (т.е. записей с наибольшими метками времени), а в случае ARMv8.3 — события-максимумы в отношении частичного порядка $G.\texttt{mo}$. Обе модели заданы в существенно разных стилях и, следовательно, имеют множество концептуальных отличий. Их необходимо преодолеть для доказательства теоремы 5. Главным отличием этих моделей является то, что обещающая модель памяти представлена операционно, в терминах шагов некоторой абстрактной машины, а модель ARMv8.3 является аксиоматической и задаёт семантику сценария поведения программы в виде графа с определёнными свойствами. Для преодоления этого отличия вводится операционная семантика обхода графа ARM-согласованного сценария поведения, шаги которой обещающая машина может непосредственно повторять (раздел 4.3). Другим существенным отличием являются те способы, которыми данные модели запрещают “плохие” сценарии. Например, такие как чтение потоком старой записи в $x$ в случае, если поток уже выполнял чтение из более новой записи в $x$. Для предотвращения этой ситуации модель ARMv8.3 использует аксиомы, которые проверяют отсутствие циклов в графе, а обещающая модель в этом случае использует фронты потоков и сообщений. Для того, чтобы обойти описанное различие между моделями, вводятся

дополнительные отношения на вершинах ARM-согласованного сценария поведения, которые тесно связаны с фронтами обещающей модели (раздел 4.4). С использованием этих идей и техники симуляции доказывается теорема 5 (раздел 4.5).

## 4.3 Обход ARM-согласованного сценария

В этом разделе сначала вводятся вспомогательные определения, затем формализуется понятие шага обхода и доказывается существование последовательности шагов, которые полностью обходят любой ARM-согласованный сценарий поведения (теорема 6).

**Определение 7.** *Конфигурацией обхода* для ARM-согласованного сценария поведения $G$ называются множества $\langle C, I\rangle$ такие, что:

- $C \subseteq G.\mathtt{E}$,
- $\mathit{dom}(G.\mathtt{po}; [C]) \subseteq C$,
- $C \cap G.\mathtt{W} \subseteq I \subseteq G.\mathtt{W}$.

Элементы $C$ мы будем называть *покрытыми* (covered) событиями, а $I$ — *выпущенными* (issued) событиями.

Покрытые события при симуляции будут соответствовать префиксу программы, который исполнен обещающей машиной, а выпущенные — записями, находящимся в памяти обещающей машины в соответствующий момент исполнения.

**Определение 8.** *Начальной конфигурацией* ARM-согласованного сценария поведения $G$ называется пара $\langle G.\mathtt{E}_0, G.\mathtt{E}_0\rangle$, где $G.\mathtt{E}_0$ — это множество инициализирующих записей.

Начальная конфигурация является конфигурацией обхода, поскольку $G.\mathtt{E}_0 \subseteq G.\mathtt{E}$, $\mathit{dom}(G.\mathtt{po}; [G.\mathtt{E}_0]) = \varnothing$ и $G.\mathtt{E}_0 \cap G.\mathtt{W} = G.\mathtt{E}_0$.

**Определение 9.** Для сценария поведения $G$ и его покрытия $C \subseteq G.\mathtt{E}$ *множеством следующих событий* $\mathsf{Next}(G, C)$ называется множество, состоящие из событий, для которых все $\mathtt{po}$-предшествующие события уже покрыты:

$$\mathsf{Next}(G, C) \triangleq \{b \in G.\mathtt{E} \mid \mathit{dom}(G.\mathtt{po}; [b]) \subseteq C\} \setminus C.$$

При симуляции данное множество будет содержать по одному событию для каждого ещё не завершённого в обещающей машине потока, причём события будут соответствовать следующему переходу в потоке — т.е. переходу, который не является обещанием. Так, если бы обещающая семантика была определена поверх некоторого синтаксиса, то множество следующих событий соответствовало бы инструкциям, на которые указывают счётчики команд потоков (program counters) в обещающей модели.

**Определение 10.** В конфигурации обхода $\langle C, I\rangle$ событие $a$ является *покрываемым* для сценария поведения $G$ и обозначается $a \in \mathsf{Coverable}(G, C, I)$, если событие $a$ является:

- событием чтения, и связанное с ним событие записи оказывается выпущенным ($a \in G.\mathtt{R}$ и $\mathtt{rf}^{-1}(a) \in I$),
- или выпущенным событием ($a \in I$),
- или барьером памяти ($a \in G.\mathtt{F}$).

В нашем обходе, заданном операционным способом, имеется шаг, который “покрывает” событие, т.е. добавляет его во множество покрытых. Событие, которое покрывается данным правилом, должно соответствовать представленным выше ограничениям. Эти ограничения, в свою очередь, соответствуют ограничениям обещающей машины. Например, обещающая машина может прочитать из некоторого конкретного сообщения только тогда, когда данное сообщение есть в памяти, т.е. соответствующее ему событие $w$ уже выпущено ($w \in I$).

**Определение 11.** Для сценария $G$ событие $w$ называется *выпускаемым* в конфигурации обхода $\langle C, I\rangle$, $w \in \mathsf{Issuable}(G, C, I)$, если выполнены следующие условия:

- $w$ является событием записи ($w \in G.\mathtt{W}$);
- все $\mathtt{po}$-предшествующие барьеры покрыты ($\mathit{dom}([\mathtt{F}]; G.\mathtt{po}; [w]) \subseteq C$); (WRITE-BOB)
- все записи других потоков, от которых зависит $w$, выпущены ($\mathit{dom}(G.\mathtt{rfe}; G.\mathtt{dob}^{+}; [w]) \subseteq I$). (WRITE-DOB)

Если шаг обхода “выпускает” событие записи, то при симуляции он соответствует обещанию, которое делает обещающая машина. Ограничение WRITE-BOB является более строгим, чем необходимо для симуляции — в обещающей модели обещание может быть сделано “через” приобретающий барьер, который соответствует $\mathsf{fence}(\mathtt{ld})$ в ARMv8.3. Тем не менее, более строгое ограничение позволяет

упростить симуляцию. Ограничение WRITE-DOB требуется для того, чтобы обещающая модель могла сертифицировать обещание, которое она делает. Шаги обхода задаются так:

$$\frac{a \in \mathsf{Next}(G, C) \cap \mathsf{Coverable}(G, C, I)}{G \vdash \langle C, I\rangle \to_{\text{TC}} \langle C \cup \{a\}, I\rangle} \qquad \frac{w \in \mathsf{Issuable}(G, C, I) \setminus I}{G \vdash \langle C, I\rangle \to_{\text{TC}} \langle C, I \cup \{w\}\rangle}$$

Здесь левое правило покрывает событие, если оно принадлежит множеству следующих за ним событий и при этом покрываемо в данной конфигурации. Правое правило выпускает событие записи, если событие выпускаемо и при этом ещё не выпущено.

Для того, чтобы использовать предложенный метод обхода в доказательстве корректности компиляции, нужно показать, что для любого ARM-согласованного сценария поведения существует его *полный обход*, т.е. обход, который покрывает все события из этого сценария.

**Теорема 6.** Для любого ARM-согласованного сценария поведения $G$ существует обход $G \vdash \langle G.\mathtt{E}_0, G.\mathtt{E}_0\rangle \to^*_{\text{TC}} \langle G.\mathtt{E}, G.\mathtt{W}\rangle$.

Доказательство теоремы основано на следующих вспомогательных леммах.

**Лемма 9.** Пусть пара множеств $\langle C, I\rangle$ является достижимой конфигурацией обхода ARM-согласованного исполнения $G$, т.е. $G \vdash \langle G.\mathtt{E}_0, G.\mathtt{E}_0\rangle \to^*_{\text{TC}} \langle C, I\rangle$. Тогда все события из множества $C$ покрываемы ($C \subseteq \mathsf{Coverable}(G,C,I)$), а события из $I$ выпускаемы ($I \subseteq \mathsf{Issuable}(G,C,I)$).

*Доказательство.* Утверждение леммы следует из определений отношений Coverable и Issuable, а также из того, что размеры множеств покрытых и выпущенных событий при обходе $G$ увеличиваются. □

**Лемма 10.** Пусть пара множеств $\langle C, I\rangle$ является достижимой конфигурацией обхода ARM-согласованного исполнения $G$, т.е. $G \vdash \langle G.\mathtt{E}_0, G.\mathtt{E}_0\rangle \to^*_{\text{TC}} \langle C, I\rangle$. Тогда все события записи, которые являются элементами множества следующих событий, являются выпускаемыми, т.е. $\mathtt{W} \cap \mathsf{Next}(G, C) \subseteq \mathsf{Issuable}(G,C,I)$.

*Доказательство.* Зафиксируем $w \in \mathtt{W} \cap \mathsf{Next}(G, C)$ и покажем, что $w \in \mathsf{Issuable}(G, C, I)$. Т.к. $w \in \mathsf{Next}(G, C)$, то все $\mathtt{po}$-предшествующие события являются покрытыми (т.е. находятся в $C$). Из этого следует, что выполняется WRITE-

вов. Кроме того, из леммы 9 следует, что все покрытые события являются покрываемыми. Таким образом, можно заключить, что выполняется WRITE-DOB, а значит, для $w$ принадлежит $\mathsf{Issuable}(G, C, I)$. □

*Доказательство теоремы 6.* Поскольку каждый конкретный ARM-согласованный сценарий поведения является конечным графом, а шаги обхода наращивают конфигурации, то для доказательства наличия полного обхода графа достаточно показать, что для каждой не конечной конфигурации существует шаг к новой конфигурации. А именно, что для любой конфигурации обхода $\langle C, I\rangle$ сценария $G$, достижимой из начальной конфигурации ($G \vdash \langle G.\mathtt{E}_0, G.\mathtt{E}_0\rangle \to^*_{\mathrm{TC}} \langle C, I\rangle$), множество покрытых событий которой не полно ($C \neq G.\mathtt{E}$), существуют такие $C'$ и $I'$, что $G \vdash \langle C, I\rangle \to_{\mathrm{TC}} \langle C', I'\rangle$. Покажем это.

Зафиксируем $\langle C, I\rangle$. Нам известно, что $\mathsf{Next}(G, C) \neq \varnothing$, т.к. $C \neq G.\mathtt{E}$. Рассмотрим два следующих случая: (i) существует элемент $\mathsf{Next}(G, C)$, который покрываем, или сначала выпускаем, а потом покрываем; (ii) все элементы в $\mathsf{Next}(G, C)$ не покрываемы и не выпускаемы (для этого случая мы показываем, что в $G$ существует событие записи, которое выпускаемо в данной конфигурации).

**Случай 1**. Зафиксируем элемент $a \in \mathsf{Next}(G, C)$. Если $a \in \mathtt{R}$ и $\mathtt{rf}^{-1}(a) \in I$, или $a \in \mathtt{F}$, или $a \in \mathtt{W} \cap I$, тогда $a$ покрываем по определению. Если $a \in \mathtt{W} \setminus I$, тогда $a$ является выпускаемым по лемме 10.

**Случай 2**. Мы предполагаем, что не существует события из $\mathsf{Next}(G, C)$, которое покрываемо или выпускаемо. Таким образом $\mathsf{Next}(G, C) \subseteq \mathtt{R}$ является следствием леммы 10 и определения покрываемого события. Кроме того, для любого чтения события $r$ из $\mathsf{Next}(G, C)$ мы знаем, что $\mathtt{rf}^{-1}(r) \notin I$. Далее покажем, что существует событие записи, которое выпускаемо в данной конфигурации. Для этого введём вспомогательное отношение $\mathtt{eord} \triangleq (\mathtt{obs} \cup \mathtt{dob} \cup \mathtt{bob})^+$, которое антирефлексивно по определению ARM-согласованности (EXTERNAL).

Мы знаем, что имеется, как минимум, одно событие (чтения) из $\mathsf{Next}(G, C)$, которое при этом не покрываемо. Это означает, что существует событие записи, которое ещё не выпущено. Выберем событие записи $w \in \mathtt{W} \setminus I$, которое является минимальным среди не выпущенных записей по отношению $\mathtt{eord}$, т.е. $\nexists w' \in \mathtt{W} \setminus I.\ \mathtt{eord}(w', w)$. Осталось показать, что событие $w$ выпускаемо.

Т.к. $w \notin \mathsf{Next}(G, C)$, то существует событие чтения $r \in \mathsf{Next}(G, C)$ такое, что $\mathtt{po}(r, w)$ и $\mathtt{rf}^{-1}(r) \notin I$. При этом $\mathtt{rf}^{-1}(r) = \mathtt{rfe}^{-1}(r)$, т.к. $C \cap \mathtt{W} \subseteq I$ и

$\forall e \in C.\ \mathit{dom}(\mathtt{po}; [e]) \in C$. Для доказательства того, что $w$ выпускаемо, требуется показать, что выполняются два следующих утверждения.

WRITE-BOB: Пусть $f \in \mathtt{F}$ и $\mathtt{po}(f, w)$. Предположим, что $f \notin C$. Тогда $\mathtt{po}(r, f)$ и $\langle \mathtt{rfe}^{-1}(r), w\rangle \in \mathtt{obs}; \mathtt{bob}^{+} \subseteq \mathtt{eord}$. Т.к. $\mathtt{rfe}^{-1}(r) \notin I$, то существует $\mathtt{eord}$-предшествующее $w$ событие записи, которое не выпущено. Это противоречит выбору $w$.

WRITE-DOB: Пусть существует событие чтения $r'$ такое, что $\mathtt{dob}^{+}(r', w)$. Если $\mathtt{rfe}^{-1}(r') \neq \bot$, то $\langle \mathtt{rfe}^{-1}(r'), w\rangle \in \mathtt{rfe}; \mathtt{dob}^{+} \subseteq \mathtt{obs}; \mathtt{dob}^{+} \subseteq \mathtt{eord}$. По определению $w$ это означает, что $\mathtt{rfe}^{-1}(r') \in I$. □

## 4.4 Аналоги фронтов для ARM-согласованного сценария

Введём вспомогательную функцию $T : \mathit{TimeMap} = \mathbb{W} \to \mathbb{N}$, сопоставляющую событиям записи в ARM-согласованном сценарии $G$ их порядковые номера в отношении $\mathtt{mo}$ так, чтобы выполнялось **correct-tmap**$(G, T)$:

$$\begin{aligned}&\textbf{correct-tmap}(G, T) \triangleq \forall w, w'.\\ &\quad (\langle w, w'\rangle \in \mathtt{mo} \Rightarrow T(w) < T(w')) \wedge\\ &\quad (w \notin \mathit{codom}(\mathtt{mo}) \Rightarrow T(w) = 0).\end{aligned}$$

Данная функция, фактически, вводит метки времени на событиях записи $G$.

Далее введём вспомогательные отношения $\mathtt{sw}$ (synchronized-with) и $\mathtt{hb}$ (happens-before):

$$\begin{aligned}\mathtt{sw} &\triangleq [\mathtt{F}(\mathtt{sy})]; \mathtt{po}; \mathtt{rf}; \mathtt{po}; [\mathtt{F}(\mathtt{ld})]\\ \mathtt{hb} &\triangleq (\mathtt{sw} \cup \mathtt{po})^{+}.\end{aligned}$$

Здесь отношение $\mathtt{sw}$ представляет пути в графе, соответствующие передаче информации о событиях записи с помощью ARM-аналогов высвобождающего ($\mathtt{F}(\mathtt{sy})$) и приобретающего ($\mathtt{F}(\mathtt{ld})$) барьеров памяти. Отношение $\mathtt{hb}$ используется для определения отношений cur-rel, acq-rel и rel-rel:

$$\begin{aligned}\text{cur-rel} &\triangleq \mathtt{rf}^{?}; \mathtt{hb},\\ \text{acq-rel} &\triangleq \mathtt{rf}^{?}; \mathtt{hb}; ([\mathtt{F}(\mathtt{sy})]; \mathtt{po}; \mathtt{rf}; \mathtt{po})^{?},\\ \text{rel-rel} &\triangleq \mathtt{rf}^{?}; \mathtt{hb}; [\mathtt{F}(\mathtt{sy})]; \mathtt{po}.\end{aligned}$$

Эти отношения вместе с функцией $T$ будут использованы в симуляции как ограничения для базового, приобретающего и высвобождающего фронтов потока в обещающей модели памяти.

Аналогичным образом отношение msg-rel $\subseteq \mathtt{W} \times \mathtt{W}$ представляет ограничение на фронт сообщения из обещающей модели:

$$\text{msg-rel} \triangleq \mathtt{rf}^?; \mathtt{hb}; [\mathtt{F(sy)}]; \mathtt{po} \cup [\mathtt{W}].$$

Так, из отношения симуляции будет следовать, что если $\langle w, w' \rangle \in$ msg-rel и события $w$ и $w'$ соответствуют сообщениям $m$ и $m'$ в памяти обещающей модели, то $m.\tau = T(w)$, $m'.\tau = T(w')$ и $m'.view(\mathtt{loc}(w)) \geqslant m.\tau$.

## 4.5 Доказательство корректности компиляции

Для доказательства теоремы 5 введём отношение симуляции $\mathcal{I}$ между конфигурациями обхода ARM-согласованного сценария $G$ ($\langle C, I \rangle$) и состояниями обещающей машины ($\langle \mathcal{TS}, M \rangle$):

$$\mathcal{I}(C, I, \mathcal{TS}, M) \triangleq \mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M) \wedge \mathcal{I}_{\textbf{mem2}}(C, I, M) \wedge \\ \mathcal{I}_{\textbf{view}}(C, \mathcal{TS}) \wedge \mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}) \wedge \mathcal{I}_{\textbf{state}}(C, \mathcal{TS}).$$

Утверждение $\mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M)$ свидетельствует, что все выпущенные события $I$ имеют аналоги в памяти обещающей машины, причём выполненные обещания соответствуют покрытым событиям записи:

$$\mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M) \triangleq \forall w \in I.\ \exists view \leqslant \text{dom-view}(\text{msg-rel}; [w]).\\ \textbf{let}\ msg \triangleq \langle \mathtt{loc}(w) : \mathtt{val_w}(w) @ T(w), view \rangle\ \textbf{in}\\ \textbf{let}\ \mathit{promises} \triangleq \mathcal{TS}(\mathtt{tid}(w)).\mathit{promises}\ \textbf{in}\\ (w \in C \Rightarrow msg \in M \setminus \mathit{promises}) \wedge (w \notin C \Rightarrow msg \in \mathit{promises}).$$

Здесь dom-view является вспомогательной функцией, которая по множеству записей строит соответствующий фронт:

$$\text{set-view}(S) \triangleq \lambda \ell.\ \max\{T(w) \mid w \in S, \mathtt{loc}(w) = \ell\}.\\ \text{dom-view}(R) \triangleq \text{set-view}(\mathit{dom}(R)).$$

Утверждение $\mathcal{I}_{\textbf{mem2}}(C, I, M)$ обозначает обратное — что для каждого сообщения из $M$ существует соответствующее ему выпущенное событие в $I$:

$$\mathcal{I}_{\textbf{mem2}}(C, I, M) \triangleq \forall \langle \ell : v @ \tau, view \rangle \in M.\ \tau \neq 0 \Rightarrow \exists w \in I.\\ \ell = \mathtt{loc}(w) \wedge v = \mathtt{val_w}(w) \wedge \tau = T(w) \wedge view \leqslant \text{dom-view}(\text{msg-rel}; [w]).$$

$\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$ утверждает, что для любого элемента множества следующих событий $\mathsf{Next}(G, C)$ фронты, представляющие связанные с ним отношениями cur-rel,

acq-rel и rel-rel события записи, больше, чем базовый, приобретающий и высвобождающие фронты соответствующего потока:

$$\begin{aligned}\mathcal{I}_{\textbf{view}}(C, \mathcal{TS}) \triangleq \forall e \in \mathsf{Next}(G, C).\\ \quad \texttt{let}\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel}\rangle \triangleq \mathcal{TS}(\mathtt{tid}(e)).\mathcal{V}\ \texttt{in}\\ \quad \mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e]) \wedge\\ \quad \mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e]) \wedge\\ \quad \mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e]).\end{aligned}$$

$\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ показывает, что все обещания, которые ещё не выполнены, имеют специальную форму фронтов, а именно, состоят из значения высвобождающего фронта соответствующего потока и метки времени самого обещания:

$$\begin{aligned}\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}) \triangleq \forall tid, \langle \ell : \_@\tau, view\rangle \in \mathcal{TS}(tid).\textit{promises}.\\ \quad view = [\ell@\tau] \sqcup \mathcal{TS}(tid).\mathcal{V}.\mathsf{rel}.\end{aligned}$$

$\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ утверждает, что для каждого потока $tid$ существует список переходов $\{t_i\}_{i\in[1..k]}$, который соответствует событиям в $tid$-подграфе сценария $G$, и текущее состояние потока $\mathcal{TS}(tid).\sigma$ получено с помощью $p$ переходов из списка, где $p$ — это количество покрытых событий $|\mathtt{E}_{tid} \cap C|$:

$$\begin{aligned}\mathcal{I}_{\textbf{state}}(C, \mathcal{TS}) \triangleq \forall tid, k = |\mathtt{E}_{tid}|.\ \exists \{\sigma_i\}_{i\in[0..k]}, \{t_i\}_{i\in[1..k]}.\\ \quad \texttt{let}\ p \triangleq |\mathtt{E}_{tid} \cap C|\ \texttt{in}\\ \quad \mathcal{TS}(tid).\sigma = \sigma_p \wedge (\forall j \in [0..k-1].\ \sigma_j \xrightarrow{\varepsilon}{}^{*} \xrightarrow{t_j} \sigma_{j+1}) \wedge\\ \quad \forall n \in [1..k], a \in \text{nth}\ ([\mathtt{E}_{tid}]; \mathtt{po}; [\mathtt{E}_{tid}])\ (n-1).\ t_{n+1} \approx \mathtt{lab}(a).\end{aligned}$$

Здесь функция nth, вызванная с аргументами $porder$ и $n$, возвращает элементы с номером $n$ из отношения частичного порядка $porder$, а предикат $t \approx \mathtt{lab}(a)$ является истинным тогда и только тогда, когда метка перехода $t$ соответствует метке события $a$. Формальные определения nth и $\approx$ приведены в приложении Г.3.

С помощью отношения симуляции показывается, что обещающая машина может сделать шаг, повторяющий шаг обхода ARM-согласованного исполнения:

**Лемма 11.** Пусть для некоторых конфигураций обхода $\langle C, I\rangle$ и $\langle C', I'\rangle$ ARM-согласованного сценария $G$, а также некоторого состояния обещающей машины $\langle \mathcal{TS}, M\rangle$ выполняется $G \vdash \langle C, I\rangle \to_{\text{TC}} \langle C', I'\rangle$ и $\mathcal{I}(C, I, \mathcal{TS}, M)$. Тогда существуют такие $\mathcal{TS}'$ и $M'$, что $\langle \mathcal{TS}, M\rangle \xrightarrow[\text{Promise}]{}{}^{+} \langle \mathcal{TS}', M'\rangle$ и $\mathcal{I}(C', I', \mathcal{TS}', M')$.

Доказательство леммы 11 приведено в приложении Д.

Основной результат данной главы, теорема 5 о корректности компиляции из обещающей модели в модель ARMv8.3, доказывается с использованием приведённой леммы.

**Теорема 5.** Для любой программы $Prog$, результата её компиляции $Prog_{\text{ARM}}$ и ARM-согласованного сценария поведения $G$ программы $Prog_{\text{ARM}}$ существует такой сценарий поведения обещающей машины, что финальное состояние памяти машины совпадает с состоянием памяти $G$.

*Доказательство.* Зафиксируем ARM-согласованный сценарий $G$. Покажем, что обещающая модель может симулировать обход сценария $G$, т.е. существуют такие $\mathcal{TS}$ и $M$, что $\mathcal{I}(G.\mathtt{E}, G.\mathtt{W}, \mathcal{TS}, M)$ и $\langle \mathcal{TS}_{\text{init}}, M_{\text{init}} \rangle \xrightarrow[\text{Promise}]{}{}^{*} \langle \mathcal{TS}, M \rangle$. Для этого проведём индукцию по полному обходу сценария $G$, существование которого следует из теоремы 6. Базой индукции в данном случае является то, что начальная конфигурация обхода сценария $G$ связана отношением $\mathcal{I}$ с начальным состоянием обещающей машины, т.е. $\mathcal{I}(G.\mathtt{E}_0, G.\mathtt{E}_0, \mathcal{TS}_{\text{init}}, M_{\text{init}})$ выполняется. Это утверждение очевидным образом следует из определения $\mathcal{I}$. Индукционным переходом в данном случае является лемма 11. Из истинности $\mathcal{I}(G.\mathtt{E}, G.\mathtt{W}, \mathcal{TS}, M)$ следует, что финальное состояние памяти машины $M$ совпадает с состоянием памяти $G$. □

## 4.6 Выводы

В этой главе приведено доказательство корректности компиляции для подмножества обещающей модели памяти [11] в модель памяти ARMv8.3 [10]. Рассмотренное подмножество состоит из ослабленных операций чтения и записи, приобретающих и высвобождающих барьеров памяти.

Доказательство базируется на новой идее обхода аксиоматических сценариев поведения, который может быть симулирован обещающей машиной. Данная идея может быть использована для доказательств корректности компиляции в другие модели памяти, что ценно, т.к. подобные задачи регулярно появляются ввиду бурного развития области. В дальнейшие планы входит расширение доказательства до полной обещающей модели памяти. Для этого нужно будет поддержать операции приобретающего чтения (read acquire) и высвобождающей записи (write release), атомарные инструкции чтения и записи (read-modify-write), частным случаем которых является сравнение с обменом (compare-and-set), а также полные барьеры памяти (SC fences). Поддержка данных инструкций потребует небольшого усложнения обхода ARM-согласованных исполнений, а также суще-

ственного усложнения отношения симуляции ввиду большого количества деталей, связанных с упомянутыми инструкциями в рамках обещающей модели памяти.

# Заключение

Основные результаты работы заключаются в следующем.

1. Разработана операционная модель памяти C/C++11. Данная модель допускает такие же сценарии поведения, что и модель C/C++11 на большинстве тестов, приведенных в литературе, но не обладает сценариями поведения со “значениями из воздуха”. В отличие от обещающей модели, предлагаемая модель является запускаемой, что упрощает разработку средств анализа программ для неё. Недостатком модели является то, она накладывает синтаксические ограничения на поведения программ.
2. Доказана корректность компиляции из существенного подмножества обещающей модели в операционную модель памяти ARMv8 POP.
3. Доказана корректность компиляции из существенного подмножества обещающей модели в аксиоматическую модель памяти ARMv8.3.

В рамках **рекомендации по применению результатов работы** в индустрии и научных исследованиях указывается, что модель памяти промышленного языка программирования должна быть лишена сценариев поведения, имеющих “значения из воздуха”, а также либо быть представленной в операционной форме, либо иметь эквивалентный операционный аналог. Последнее позволяет реализовать интерпретатор модели и выполнять отладку программ в рамках модели.

Также были определены **перспективы дальнейшей разработки тематики**, основным из которых является разработка обобщенной аксиоматической модели памяти для процессорных архитектур, которая будет определена для синтаксиса модели C/C++11 и окажется строгим надмножеством существующих моделей памяти x86, Power и ARM, а также для которой будет применим предложенный метод доказательства корректности компиляции из обещающей модели памяти. Это позволит свести дальнейшие доказательства корректности компиляции из обещающей модели к доказательству корректности компиляции в обобщенную аксиоматическую модель, что сводится к рассуждениям об ацикличности и вложенности путей на графах. Кроме того, актуальной является задача разработки эффективной программной логики на базе логики многопоточного разделения (concurrent separation logic) для операционного аналога модели памяти C/C++11 и обещающей модели памяти. Такая логика позволит формально доказывать в рамках моделей сложные свойства программ, такие как соответствие спецификации.

# Список сокращений и условных обозначений

| | | Стр. |
|---|---|---|
| **CAS** | compare-and-set, инструкция атомарного сравнения и записи, частный случай **RMW** | 51 |
| **C/C++11 MM** | C/C++11 memory model, модель памяти C/C++11 | 24 |
| **DR** | data race, гонка по данным | 40 |
| **JMM** | Java memory model, модель памяти Java | 21 |
| **LB** | load buffering, шаблон “буферизация чтения” | 19 |
| **MM** | memory model, модель памяти | 15 |
| **MP** | message passing, шаблон “передача сообщения” | 13 |
| **OOTA** | out-of-thin-air values, “значения из воздуха” | 12 |
| OpC11 **MM** | операционная модель памяти для C/C++11, предложенная диссертантом | 33 |
| **POP** | partial order propagation, подсистема памяти модели ARMv8 POP, основанная на частичных порядках | 73 |
| Promise **MM** | обещающая модель памяти | 69 |
| **QSBR** | quiescent state-based reclamation, стратегия “освобождение памяти в момент затишья” | 62 |
| **RCU** | read-copy-update, структура данных, поддерживающая неблокирующую синхронизацию для ситуации “один писатель много читателей” | 62 |
| **RMW** | read-modify-write, атомарное чтение-запись | 24 |
| **SB** | store buffering, шаблон “буферизация записи” | 40 |
| **SC** | sequential consistency, модель последовательной консистентности | 12 |
| **SE** | speculative execution, шаблон “спекулятивное исполнение” | 37 |

# Список литературы


1. Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs / L. Lamport // IEEE Trans. Computers. — 1979. — Vol. 28, no. 9. — P. 690–691.

2. Adve, S. V. Shared Memory Consistency Models: A Tutorial / S. V. Adve, K. Gharachorloo // IEEE Computer. — 1996. — Vol. 29, no. 12. — P. 66–76.

3. Manson, J. The Java Memory Model / J. Manson, W. Pugh, S. V. Adve // Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005. — ACM, 2005. — P. 378–391.

4. Batty, M. Mathematizing C++ Concurrency / M. Batty, S. Owens, S. Sarkar, P. Sewell, T. Weber // Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011. — ACM, 2011. — P. 55–66.

5. Boehm, H.-J. Outlawing Ghosts: Avoiding Out-of-thin-air Results / H.-J. Boehm, B. Demsky // MSPC 2014. — ACM, 2014. — P. 7:1–7:6.

6. Alglave, J. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory / J. Alglave, L. Maranget, M. Tautschnig // ACM Trans. Program. Lang. Syst. — 2014. — Vol. 36, no. 2. — P. 7:1–7:74.

7. Sewell, P. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors / P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, M. O. Myreen // Commun. ACM. — 2010. — Vol. 53, no. 7. — P. 89–97.

8. Owens, S. A Better x86 Memory Model: x86-TSO / S. Owens, S. Sarkar, P. Sewell // Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. — 2009. — P. 391–407.

9. Flur, S. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA / S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon,

P. Sewell // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. — ACM, 2016. — P. 608–621.

10. Pulte, C. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8 / C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, P. Sewell // POPL 2018. — 2018.

11. Kang, J. A Promising Semantics for Relaxed-Memory Concurrency / J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. — ACM, 2017.

12. Keller, R. M. Formal Verification of Parallel Programs / R. M. Keller // Commun. ACM. — 1976. — Vol. 19, no. 7. — P. 371–384.

13. Felleisen, M. The Revised Report on the Syntactic Theories of Sequential Control and State / M. Felleisen, R. Hieb // Theor. Comput. Sci. — 1992. — Vol. 103, no. 2. — P. 235–271.

14. Lynch, N. A. Forward and Backward Simulations: I. Untimed Systems / N. A. Lynch, F. W. Vaandrager // Inf. Comput. — 1995. — Vol. 121, no. 2. — P. 214–233.

15. Lynch, N. A. Forward and Backward Simulations, II: Timing-Based Systems / N. A. Lynch, F. W. Vaandrager // Inf. Comput. — 1996. — Vol. 128, no. 1. — P. 1–25.

16. Flatt, M. — Reference: Racket. PLT-TR-2010-1 [Электронный ресурс]. — URL: https://racket-lang.org/tr1/ (дата обращения: 29.12.2017).

17. Язык программирования Racket [Электронный ресурс]. — URL: http://racket-lang.org/ (дата обращения: 29.12.2017).

18. Felleisen, M. Semantics Engineering with PLT Redex / M. Felleisen, R. B. Findler, M. Flatt. — MIT Press, 2009.

19. Run your research: on the effectiveness of lightweight mechanization / C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy,

J. Rafkind, S. Tobin-Hochstadt, R. B. Findler // Proceedings of the 39th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Philadelphia, Pennsylvania, USA, January, 2012. — ACM, 2012. — P. 285–296.

20. Lahav, O. Explaining Relaxed Memory Models with Program Transformations / O. Lahav, V. Vafeiadis // FM 2016: Formal Methods — 21st International Symposium, Limassol, Cyprus, November 9–11, 2016, Proceedings. — Springer, 2016.

21. Подкопаев, А. О корректности компиляции подмножества обещающей модели памяти в аксиоматическую модель ARMv8.3 / А. Подкопаев, О. Лахав, В. Вафеядис // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. — 2017. — Т. 10, № 4. — С. 51–69.

22. Подкопаев, А. Обещающая компиляция в ARMv8.3 / А. Подкопаев, О. Лахав, В. Вафеядис // Труды ИСП РАН. — 2017. — Т. 29, № 5. — С. 149–164.

23. Podkopaev, A. Promising Compilation to ARMv8 POP / A. Podkopaev, O. Lahav, V. Vafeiadis // Leibniz International Proceedings in Informatics (LIPIcs), 31st European Conference on Object-Oriented Programming (ECOOP 2017) / Ed. by Peter Müller. — Vol. 74. — Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. — P. 22:1–22:28. — URL: http://drops.dagstuhl.de/opus/volltexte/2017/7266.

24. Подкопаев, А. Обещающая компиляция в ARMv8 / А. Подкопаев, О. Лахав, В. Вафеядис // Труды конференции, Языки программирования и компиляторы / Под ред. Д.В. Дубров. — Труды конференции. — Ростов-на-Дону, Россия: 2017.

25. Podkopaev, A. Operational Aspects of C/C++ Concurrency / A. Podkopaev, I. Sergey, A. Nanevski // CoRR. — 2016. — Vol. abs/1606.01400. — URL: http://arxiv.org/abs/1606.01400.

26. Kshemkalyani, A. D. Distributed computing: principles, algorithms, and systems / A. D. Kshemkalyani, M. Singhal. — Cambridge University Press, 2011.

27. Hennessy, J. L. Computer Architecture — A Quantitative Approach (5. ed.) / J. L. Hennessy, D. A. Patterson. — Morgan Kaufmann, 2012.

28. Aho, A. V. Compilers: Principles, Techniques, and Tools / A. V. Aho, R. Sethi, J. D. Ullman. Addison-Wesley series in computer science / World student series edition. — Addison-Wesley, 1986.

29. Muchnick, S. S. Advanced Compiler Design and Implementation / S. S. Muchnick. — Morgan Kaufmann, 1997.

30. ISO/IEC 9899:2011. Programming language C. — 2011.

31. ISO/IEC 14882:2011. Programming language C++. — 2011.

32. Unger, S. H. Hazards, Critical Races, and Metastability / S. H. Unger // IEEE Trans. Computers. — 1995. — Vol. 44, no. 6. — P. 754–768.

33. Sarkar, S. Understanding POWER Multiprocessors / S. Sarkar, P. Sewell, J. Alglave, L. Maranget, D. Williams // Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011. — ACM, 2011. — P. 175–186.

34. Kavanagh, R. A Denotational Semantics for SPARC TSO / R. Kavanagh, S. Brookes // CoRR. — 2017. — Vol. abs/1711.00931. — URL: http://arxiv.org/abs/1711.00931.

35. Crary, K. A Calculus for Relaxed Memory / K. Crary, M. J. Sullivan // Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15–17, 2015. — 2015. — P. 623–636.

36. Boudol, G. Relaxed Operational Semantics of Concurrent Programming Languages / G. Boudol, G. Petri, B. P. Serpette // EXPRESS 2012. — 2012. — P. 19–33.

37. Boudol, G. Relaxed memory models: an operational approach / G. Boudol, G. Petri // POPL 2009. — 2009. — P. 392–403.

38. Pichon-Pharabod, J. A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions / J. Pichon-Pharabod, P. Sewell // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. — ACM, 2016. — P. 622–633.

39. Jeffrey, A. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory / A. Jeffrey, J. Riely // Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5–8, 2016. — 2016. — P. 759–767.

40. Nienhuis, K. An operational semantics for C/C++11 concurrency / K. Nienhuis, K. Memarian, P. Sewell // Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 – November 4, 2016. — 2016. — P. 111–128.

41. Diehl, S. Abstract machines for programming language implementation / S. Diehl, P. H. Hartel, P. Sestoft // Future Generation Comp. Syst. — 2000. — Vol. 16, no. 7. — P. 739–751.

42. Vafeiadis, V. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it / V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, F. Zappa Nardelli // Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15–17, 2015. — 2015. — P. 209–220.

43. Morisset, R. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model / R. Morisset, P. Pawan, F. Zappa Nardelli // ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013. — 2013. — P. 187–196.

44. Sevcík, J. On Validity of Program Transformations in the Java Memory Model / J. Sevcík, D. Aspinall // ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings. — 2008. — P. 27–51.

45. Peled, D. A. Model Checking / D. A. Peled, P. Pelliccione, P. Spoletini // Wiley Encyclopedia of Computer Science and Engineering. — 2008.

46. Clarke, E. M. Model checking / E. M. Clarke, O. Grumberg, D. Peled. — MIT press, 1999.

47. Hoare, C. A. R. An Axiomatic Basis for Computer Programming / C. A. R. Hoare // Commun. ACM. — 1969. — Vol. 12, no. 10. — P. 576–580.

48. Owicki, S. S. An Axiomatic Proof Technique for Parallel Programs I / S. S. Owicki, D. Gries // Acta Inf. — 1976. — Vol. 6. — P. 319–340.

49. Owicki, S. S. Verifying Properties of Parallel Programs: An Axiomatic Approach / S. S. Owicki, D. Gries // Commun. ACM. — 1976. — Vol. 19, no. 5. — P. 279–285.

50. O'Hearn, P. W. Resources, Concurrency and Local Reasoning / P. W. O'Hearn // CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings. — 2004. — P. 49–67.

51. Bornat, R. Permission accounting in separation logic / R. Bornat, C. Calcagno, P. W. O'Hearn, M. J. Parkinson // Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005. — 2005. — P. 259–270.

52. Hobor, A. Oracle Semantics for Concurrent Separation Logic / A. Hobor, A. W. Appel, F. Zappa Nardelli // Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings. — 2008. — P. 353–367.

53. Dockins, R. Multimodal Separation Logic for Reasoning About Operational Semantics / R. Dockins, A. W. Appel, A. Hobor // Electr. Notes Theor. Comput. Sci. — 2008. — Vol. 218. — P. 5–20.

54. Hobor, A. Barriers in Concurrent Separation Logic / A. Hobor, C. Gherghina // Programming Languages and Systems — 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings. — 2011. — P. 276–296.

55. Gotsman, A. Local Reasoning for Storable Locks and Threads / A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv // Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings. — 2007. — P. 19–37.

56. Jacobs, B. Expressive modular fine-grained concurrency specification / B. Jacobs, F. Piessens // Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on

Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011. — 2011. — P. 271–282.

57. Svendsen, K. Impredicative Concurrent Abstract Predicates / K. Svendsen, L. Birkedal // Programming Languages and Systems — 23rd European Symposium on Programming, ESOP 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. 149–168.

58. Dinsdale-Young, T. Concurrent Abstract Predicates / T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, V. Vafeiadis // ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21–25, 2010. Proceedings. — 2010. — P. 504–528.

59. Jung, R. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning / R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, D. Dreyer // Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15–17, 2015. — 2015. — P. 637–650.

60. Ley-Wild, R. Subjective auxiliary state for coarse-grained concurrency / R. Ley-Wild, A. Nanevski // The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy, January 23–25, 2013. — 2013. — P. 561–574.

61. Sergey, I. Mechanized verification of fine-grained concurrent programs / I. Sergey, A. Nanevski, A. Banerjee // Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015. — 2015. — P. 77–87.

62. Vechev, M. T. Abstraction-guided synthesis of synchronization / M. T. Vechev, E. Yahav, G. Yorsh // STTT. — 2013. — Vol. 15, no. 5-6. — P. 413–431.

63. Raychev, V. Automatic Synthesis of Deterministic Concurrency / V. Raychev, M. T. Vechev, E. Yahav // Static Analysis — 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013. Proceedings. — 2013. — P. 283–303.

64. Kaiser, J.-O. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris / J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, V. Vafeiadis //

31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain. — 2017. — P. 17:1–17:29.

65. Vafeiadis, V. Relaxed separation logic: a program logic for C11 concurrency / V. Vafeiadis, C. Narayan // Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013. — 2013. — P. 867–884.

66. Turon, A. GPS: navigating weak memory with ghosts, protocols, and separation / A. Turon, V. Vafeiadis, D. Dreyer // Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20–24, 2014. — 2014. — P. 691–707.

67. Lahav, O. Owicki-Gries Reasoning for Weak Memory Models / O. Lahav, V. Vafeiadis // Automata, Languages, and Programming — 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II. — 2015. — P. 311–323.

68. Meshman, Y. Pattern-based Synthesis of Synchronization for the C++ Memory Model / Y. Meshman, N. Rinetzky, E. Yahav // Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. — 2015. — P. 120–127.

69. Dan, A. M. Predicate Abstraction for Relaxed Memory Models / A. M. Dan, Y. Meshman, M. T. Vechev, E. Yahav // Static Analysis — 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013. Proceedings. — 2013. — P. 84–104.

70. Документация архитектуры DEC Alpha [Электронный ресурс]. — URL: http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=alphaahb.pdf (дата обращения: 29.12.2017).

71. Batty, M. The Problem of Programming Language Concurrency Semantics / M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, P. Sewell // LNCS, Programming Languages and Systems — 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and

Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings. — Vol. 9032. — Springer, 2015. — P. 283–307.

72. Gosling, J. The Java Language Specification / J. Gosling, W. N. Joy, G. L. Steele Jr. — Addison-Wesley, 1996.

73. The Java Language Specification, Java SE 8 Edition [Электронный ресурс]. — URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html. (дата обращения: 29.12.2017).

74. Jackson, D. Software Abstractions: Logic, Language, and Analysis / D. Jackson. — The MIT Press, 2006.

75. Torlak, E. Growing Solver-aided Languages with Rosette / E. Torlak, R. Bodik // Onward! 2013, Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. — Onward! 2013. — 2013. — P. 135–152.

76. Torlak, E. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages / E. Torlak, R. Bodik // PLDI '14, Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '14. — 2014. — P. 530–541.

77. Wickerson, J. Automatically Comparing Memory Consistency Models / J. Wickerson, M. Batty, T. Sorensen, G. A. Constantinides // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. — ACM, 2017.

78. Bornholt, J. Synthesizing memory models from framework sketches and Litmus tests / J. Bornholt, E. Torlak // Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017. — 2017. — P. 467–481.

79. Memarian, K. Into the Depths of C: Elaborating the De Facto Standards / K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, P. Sewell // PLDI '16, Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '16. — 2016. — P. 1–15.

80. Pugh, W. Fixing the Java Memory Model / W. Pugh // Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12–14, 1999. — 1999. — P. 89–98.

81. Pugh, W. The Java memory model is fatally flawed / W. Pugh // Concurrency - Practice and Experience. — 2000. — Vol. 12, no. 6. — P. 445–455.

82. ISO/IEC 14882:2014. Programming language C++. — 2014.

83. ISO/IEC 14882:2017. Programming language C++. — 2017.

84. Batty, M. Clarifying and compiling C/C++ concurrency: from C++11 to POWER / M. Batty, K. Memarian, S. Owens, S. Sarkar, P. Sewell // POPL 2012. — 2012. — P. 509–520.

85. Chakraborty, S. Validating optimizations of concurrent C/C++ programs / S. Chakraborty, V. Vafeiadis // CGO 2016. — 2016. — P. 216–226.

86. Lahav, O. Repairing sequential consistency in C/C++11 / O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, D. Dreyer // PLDI 2017. — 2017. — P. 618–632.

87. Batty, M. Overhauling SC atomics in C11 and OpenCL / M. Batty, A. F. Donaldson, J. Wickerson // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. — 2016. — P. 634–648.

88. Doko, M. A Program Logic for C11 Memory Fences / M. Doko, V. Vafeiadis // Verification, Model Checking, and Abstract Interpretation — 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings. — 2016. — P. 413–430.

89. Vafeiadis, V. Formal Reasoning about the C11 Weak Memory Model / V. Vafeiadis // CPP 2015. — 2015. — P. 1–2.

90. Batty, M. Library abstraction for C/C++ concurrency / M. Batty, M. Dodds, A. Gotsman // POPL 2013. — 2013. — P. 235–248.

91. Lidbury, C. Dynamic race detection for C++11 / C. Lidbury, A. F. Donaldson // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. — 2017. — P. 443–457.

92. Tassarotti, J. Verifying read-copy-update in a logic for weak memory / J. Tassarotti, D. Dreyer, V. Vafeiadis // Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015. — 2015. — P. 110–120.

93. Doko, M. Tackling Real-Life Relaxed Concurrency with FSL++ / M. Doko, V. Vafeiadis // Programming Languages and Systems — 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings. — 2017. — P. 448–475.

94. Batty, M. — The Thin-air Problem [Электронный ресурс]. — URL: http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html (дата обращения: 29.12.2017).

95. Winskel, G. Event Structures / G. Winskel // Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986. — 1986. — P. 325–392.

96. Winskel, G. An introduction to event structures / G. Winskel // Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 – June 3, 1988, Proceedings. — 1988. — P. 364–397.

97. Трифанов, В. Динамическое обнаружение состояний гонки в многопоточных Java-программах: дис. канд. техн. наук: 05.13.11 / В. Трифанов. — СПб, 2013. — 112 с.

98. Подкопаев, А. — Интерпретатор для операционной модели C/++11 [Электронный ресурс]. — URL: https://github.com/anlun/OperationalSemanticsC11 (дата обращения: 29.12.2017).

99. Bornat, R. New Lace and Arsenic: adventures in weak memory with a program logic / R. Bornat, J. Alglave, M. J. Parkinson // CoRR. — 2015. — Vol. abs/1512.01416. — URL: http://arxiv.org/abs/1512.01416.

100. Lahav, O. Taming release-acquire consistency / O. Lahav, N. Giannarakis, V. Vafeiadis // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016. — 2016. — P. 649–662.

101. Maranget, L. — 2012. — A Tutorial Introduction to the ARM and POWER Relaxed Memory Models [Электронный ресурс]. — URL: http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf (дата обращения: 29.12.2017).

102. Семейство компиляторов GCC [Электронный ресурс]. — URL: https://gcc.gnu.org/ (дата обращения: 29.12.2017).

103. Компилятор языка C в платформу LLVM [Электронный ресурс]. — URL: https://clang.llvm.org/ (дата обращения: 29.12.2017).

104. McKenney, P. E. Read-Copy Update: Using Execution History to Solve Concurrency Problems / P. E. McKenney, J. D. Slingwine // PDCS. — 1998. — P. 509–518.

105. McKenney, P. E. Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels: Ph.D. thesis / OGI School of Science and Engineering at Oregon Health and Sciences University. — 2004.

106. Desnoyers, M. User-Level Implementations of Read-Copy Update / M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, J. Walpole // IEEE Transactions on Parallel and Distributed Systems. — 2012. — Feb. — Vol. 23, no. 2. — P. 375–382.

107. Hritcu, C. Testing noninterference, quickly / C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, L. Lampropoulos // Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013). — 2013. — P. 455–468.

108. Milner, R. Communication and concurrency / R. Milner. PHI Series in computer science. — Prentice Hall, 1989.

109. ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile [Электронный ресурс]. — URL: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile (дата обращения: 29.12.2017).

# Список рисунков

# Список таблиц

# Приложение А. Каталог тестов для модели C/C++11

## А.1 Буферизация записи, SB

**SB-rel-acq**

$$\begin{array}{l|l} [x] :=_{\texttt{rel}} 0; & [y] :=_{\texttt{rel}} 0; \\ [x] :=_{\texttt{rel}} 1; & [y] :=_{\texttt{rel}} 1; \\ a :=_{\texttt{acq}} [y] & b :=_{\texttt{acq}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**SB-sc**

$$\begin{array}{l|l} [x] :=_{\texttt{sc}} 0; & [y] :=_{\texttt{sc}} 0; \\ [x] :=_{\texttt{sc}} 1; & [y] :=_{\texttt{sc}} 1; \\ a :=_{\texttt{sc}} [y] & b :=_{\texttt{sc}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**SB-sc-rel**

$$\begin{array}{l|l} [x] :=_{\texttt{sc}} 0; & [y] :=_{\texttt{sc}} 0; \\ [x] :=_{\texttt{rel}} 1; & [y] :=_{\texttt{sc}} 1; \\ a :=_{\texttt{sc}} [y] & b :=_{\texttt{sc}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**SB-sc-acq**

$$\begin{array}{c|c} [x] :=_{\texttt{sc}} 0; & [y] :=_{\texttt{sc}} 0; \\ [x] :=_{\texttt{sc}} 1; & [y] :=_{\texttt{sc}} 1; \\ a :=_{\texttt{acq}} [y] & b :=_{\texttt{sc}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

## A.2 Буферизация чтения, LB

**LB-rlx**

$$\begin{array}{c|c} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y] & b :=_{\texttt{rlx}} [x] \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-rel-rlx**

$$\begin{array}{c|c} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y] & b :=_{\texttt{rlx}} [x] \\ [x] :=_{\texttt{rel}} 1; & [y] :=_{\texttt{rel}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-acq-rlx**

$$\begin{array}{c|c} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{acq}} [y] & b :=_{\texttt{acq}} [x] \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-rel-acq-rlx**

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{acq}} [y] & b :=_{\texttt{rlx}} [x] \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rel}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0].$$

---

**LB-rlx-use**

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y] & b :=_{\texttt{rlx}} [x] \\ [z1] :=_{\texttt{rlx}} a; & [z2] :=_{\texttt{rlx}} b; \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-rlx-let**

$$\begin{array}{l|l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y] & b :=_{\texttt{rlx}} [x] \\ a' := a + 1; & b' := b + 1; \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, a' = 1, b = 0, b' = 1], [a = 0, a' = 1, b = 1, b' = 2],$$

$$[a = 1, a' = 2, b = 0, b' = 1], [a = 1, a' = 2, b = 1, b' = 2].$$

---

**LB-rlx-join**

$$\begin{array}{ccc|c|ccc} & & & [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; & & \\ a :=_{\texttt{rlx}} [y]; & & & \| & b :=_{\texttt{rlx}} [x]; & & \\ [z1] :=_{\texttt{rlx}} a & \| & \textbf{skip} & \| & [z2] :=_{\texttt{rlx}} b & \| & \textbf{skip} \\ [x] :=_{\texttt{rlx}} 1 & & & \| & [y] :=_{\texttt{rlx}} 1 & & \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-rel-rlx-join**

$$\begin{array}{ccc|c|ccc} & & & [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; & & \\ a :=_{\texttt{rlx}} [y]; & & & \| & b :=_{\texttt{rlx}} [x]; & & \\ [z1] :=_{\texttt{rlx}} a & \| & \textbf{skip} & \| & [z2] :=_{\texttt{rlx}} b & \| & \textbf{skip} \\ [x] :=_{\texttt{rel}} 1 & & & \| & [y] :=_{\texttt{rel}} 1 & & \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

---

**LB-acq-rlx-join**

$$\begin{array}{ccc|c|ccc} & & & [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; & & \\ a :=_{\texttt{acq}} [y]; & & & \| & b :=_{\texttt{acq}} [x]; & & \\ [z1] :=_{\texttt{rlx}} a & \| & \textbf{skip} & \| & [z2] :=_{\texttt{rlx}} b & \| & \textbf{skip} \\ [x] :=_{\texttt{rlx}} 1 & & & \| & [y] :=_{\texttt{rlx}} 1 & & \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 0, b = 1], [a = 1, b = 0], [a = 1, b = 1].$$

## A.3 Передача сообщения, MP

**MP-rlx-na**

$$\begin{array}{l} [f] :=_{\texttt{rlx}} 0; [d] :=_{\texttt{na}} 0; \\ [d] :=_{\texttt{na}} 5; \\ [f] :=_{\texttt{rlx}} 1 \end{array} \Bigg\| \begin{array}{l} \textbf{repeat}\, [f]_{\texttt{rlx}}\, \textbf{end}; \\ a :=_{\texttt{na}} [d] \end{array}$$

Результаты в C/C++11 MM:

**undefined behavior**.

---

**MP-rel-rlx-na**

$$\begin{array}{l} [f] :=_{\texttt{rlx}} 0; [d] :=_{\texttt{na}} 0; \\ [d] :=_{\texttt{na}} 5; \\ [f] :=_{\texttt{rel}} 1 \end{array} \Bigg\| \begin{array}{l} \textbf{repeat}\, [f]_{\texttt{rlx}}\, \textbf{end}; \\ a :=_{\texttt{na}} [d] \end{array}$$

Результаты в C/C++11 MM:

**undefined behavior**.

---

**MP-rlx-acq-na**

$$\begin{array}{l} [f] :=_{\texttt{rlx}} 0; [d] :=_{\texttt{na}} 0; \\ [d] :=_{\texttt{na}} 5; \\ [f] :=_{\texttt{rlx}} 1 \end{array} \Bigg\| \begin{array}{l} \textbf{repeat}\, [f]_{\texttt{acq}}\, \textbf{end}; \\ a :=_{\texttt{na}} [d] \end{array}$$

Результаты в C/C++11 MM:

**undefined behavior**.

---

**MP-rel-acq-na**

$$\begin{array}{l} [f] :=_{\texttt{rlx}} 0; [d] :=_{\texttt{na}} 0; \\ [d] :=_{\texttt{na}} 5; \\ [f] :=_{\texttt{rel}} 1 \end{array} \Bigg\| \begin{array}{l} \textbf{repeat}\, [f]_{\texttt{acq}}\, \textbf{end}; \\ a :=_{\texttt{na}} [d] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 5].$$

---

**MP-rel-acq-na-rlx**

$$
\begin{array}{l||l}
[f] :=_{\texttt{rel}} 0; [d] :=_{\texttt{na}} 0; & \\
[d] :=_{\texttt{na}} 5; & \textbf{repeat}\ [f]_{\texttt{acq}} == 2\ \textbf{end}; \\
[f] :=_{\texttt{rel}} 1; & a :=_{\texttt{na}} [d] \\
[f] :=_{\texttt{rlx}} 2 &
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 5].$$

---

**MP-rel-acq-na-rlx_2**

$$
\begin{array}{l||l}
[f] :=_{\texttt{na}} 0; [d] :=_{\texttt{na}} 0; [x] :=_{\texttt{na}} 0; & \\
[d] :=_{\texttt{na}} 5; & \textbf{repeat}\ [f]_{\texttt{acq}} == 2\ \textbf{end}; \\
[f] :=_{\texttt{rel}} 1; & a :=_{\texttt{na}} [d] \\
[x] :=_{\texttt{rel}} 1; & b :=_{\texttt{rlx}} [x] \\
[f] :=_{\texttt{rlx}} 2 &
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 5, b = 0], [a = 5, b = 1].$$

---

**MP-con-na**

$$
\begin{array}{l||l}
[f] :=_{\texttt{rlx}} \textbf{null}; [d] :=_{\texttt{na}} 0; & \\
[d] :=_{\texttt{na}} 5; & a :=_{\texttt{con}} [f]; \\
 & \textbf{if}\ a \neq \textbf{null} \\
[f] :=_{\texttt{rel}} d & \textbf{then}\ b :=_{\texttt{na}} [a] \\
 & \textbf{else}\ b := 0 \\
 & \textbf{fi}
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = \textbf{null}, b = 0], [a = d, b = 5].$$

---

**MP-con-na_2**

$$
\begin{array}{l||l}
[p] :=_{\texttt{na}} \textbf{null}; [d] :=_{\texttt{na}} 0; [x] :=_{\texttt{na}} 0; & \\
\hline
[x] :=_{\texttt{rlx}} 1; & a :=_{\texttt{con}} [p]; \\
[d] :=_{\texttt{na}} 1; & \textbf{if } a \neq \textbf{null} \\
[p] :=_{\texttt{rel}} d & \textbf{then } b :=_{\texttt{na}} [a]; \\
 & \quad\; c :=_{\texttt{rlx}} [x] \\
 & \textbf{else } b := 0; c := 0 \\
 & \textbf{fi}
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = \textbf{null}, b = 0, c = 0], [a = d, b = 5, c = 0], [a = d, b = 5, c = 1].$$

---

**MP-cas-rel-acq-na**

$$
\begin{array}{l||l||l}
 & [f] :=_{\texttt{rlx}} 1; [d] :=_{\texttt{na}} 0; & \\
\hline
[d] :=_{\texttt{na}} 5; & a := \textbf{cas}_{\texttt{acq,rlx}}(f, 0, 1); & b := \textbf{cas}_{\texttt{acq,rlx}}(f, 0, 1); \\
[f] :=_{\texttt{rel}} 0 & \textbf{if } a == 0 & \textbf{if } b == 0 \\
 & \textbf{then } [d] :=_{\texttt{rlx}} 6 & \textbf{then } [d] :=_{\texttt{rlx}} 7 \\
 & \textbf{else } 0 & \textbf{else } 0 \\
 & \textbf{fi} & \textbf{fi}
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 0, b = 1], [a = 1, b = 0].$$

---

**MP-cas-rel-rlx-na**

$$
\begin{array}{l||l||l}
 & [f] :=_{\texttt{rlx}} 1; [d] :=_{\texttt{na}} 0; & \\
\hline
[d] :=_{\texttt{na}} 5; & a := \textbf{cas}_{\texttt{rlx,rlx}}(f, 0, 1); & b := \textbf{cas}_{\texttt{rlx,rlx}}(f, 0, 1); \\
[f] :=_{\texttt{rel}} 0 & \textbf{if } a == 0 & \textbf{if } b == 0 \\
 & \textbf{then } [d] :=_{\texttt{rlx}} 6 & \textbf{then } [d] :=_{\texttt{rlx}} 7 \\
 & \textbf{else } 0 & \textbf{else } 0 \\
 & \textbf{fi} & \textbf{fi}
\end{array}
$$

Результаты в C/C++11 MM:

**undefined behavior**.

## A.4 Корректность повторного чтения, CoRR

**CoRR-rlx**

$$[x] :=_{\texttt{rlx}} 0;$$
$$[x] :=_{\texttt{rlx}} 1 \;\Big\|\; [x] :=_{\texttt{rlx}} 2 \;\Big\|\; \begin{array}{l} a :=_{\texttt{rlx}} [x]; \\ b :=_{\texttt{rlx}} [x] \end{array} \;\Big\|\; \begin{array}{l} c :=_{\texttt{rlx}} [x]; \\ d :=_{\texttt{rlx}} [x] \end{array}$$

В C/C++11 MM возможны все результаты, соответствующие $\{a, b, c, d\} \subseteq \{0, 1, 2\}$, кроме тех, в которых $a > b = 0$ или $c > d = 0$, а также

$$[a = 1, b = 2, c = 2, d = 1], [a = 2, b = 1, c = 1, d = 2].$$

---

**CoRR-rel-acq**

$$[x] :=_{\texttt{rel}} 0;$$
$$[x] :=_{\texttt{rel}} 1 \;\Big\|\; [x] :=_{\texttt{rel}} 2 \;\Big\|\; \begin{array}{l} a :=_{\texttt{acq}} [x]; \\ b :=_{\texttt{acq}} [x] \end{array} \;\Big\|\; \begin{array}{l} c :=_{\texttt{acq}} [x]; \\ d :=_{\texttt{acq}} [x] \end{array}$$

В C/C++11 MM возможны все результаты, соответствующие $\{a, b, c, d\} \subseteq \{0, 1, 2\}$, кроме тех, в которых $a > b = 0$ или $c > d = 0$, а также

$$[a = 1, b = 2, c = 2, d = 1], [a = 2, b = 1, c = 1, d = 2].$$

## A.5 Независимые чтения независимых записей, IRIW

**IRIW-rlx**

$$[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0;$$
$$[x] :=_{\texttt{rlx}} 1 \;\Big\|\; [y] :=_{\texttt{rlx}} 1 \;\Big\|\; \begin{array}{l} a :=_{\texttt{rlx}} [x]; \\ b :=_{\texttt{rlx}} [y] \end{array} \;\Big\|\; \begin{array}{l} c :=_{\texttt{rlx}} [y]; \\ d :=_{\texttt{rlx}} [x] \end{array}$$

В C/C++11 MM возможны все результаты, соответствующие $\{a, b, c, d\} \subseteq \{0, 1\}$.

**IRIW-rel-acq**

$$\begin{array}{ccccc} & & [x] :=_{\texttt{rel}} 0; & [y] :=_{\texttt{rel}} 0; & \\ [x] :=_{\texttt{rel}} 1 & \| & [y] :=_{\texttt{rel}} 1 & \| & \begin{array}{l} a :=_{\texttt{acq}} [x]; \\ b :=_{\texttt{acq}} [y] \end{array} \| \begin{array}{l} c :=_{\texttt{acq}} [y]; \\ d :=_{\texttt{acq}} [x] \end{array} \end{array}$$

В C/C++11 MM возможны все результаты, соответствующие $\{a, b, c, d\} \subseteq \{0, 1\}$.

**IRIW-sc**

$$\begin{array}{ccccc} & & [x] := 0; & [y] := 0; & \\ [x] := 1 & \| & [y] := 1 & \| & \begin{array}{l} a := [x]; \\ b := [y] \end{array} \| \begin{array}{l} c := [y]; \\ d := [x] \end{array} \end{array}$$

В C/C++11 MM возможны все результаты, соответствующие $\{a, b, c, d\} \subseteq \{0, 1\}$, кроме $[a = 1, b = 0, c = 1, d = 0]$.

## A.6 Зависимость запись-чтение, WRC

**WRC-rel-acq**

$$\begin{array}{ccccc} & & [x] :=_{\texttt{rel}} 0; & [y] :=_{\texttt{rel}} 0; & \\ [x] :=_{\texttt{rel}} 1 & \| & a :=_{\texttt{acq}} [x]; & \| & \begin{array}{l} b :=_{\texttt{acq}} [y]; \\ c :=_{\texttt{acq}} [x] \end{array} \\ & & [y] :=_{\texttt{rel}} a & & \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 1, b = 0, c = 0],$$

$$[a = 1, b = 0, c = 1], [a = 1, b = 1, c = 1].$$

**WRC-rlx**

$$\begin{array}{c|c|c} & [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; & \\ [x] :=_{\texttt{rlx}} 1 & a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y]; \\ & [y] :=_{\texttt{rlx}} a & c :=_{\texttt{rlx}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 1, b = 0, c = 0],$$

$$[a = 1, b = 0, c = 1], [a = 1, b = 1, c = 1],$$

$$[a = 1, b = 1, c = 0].$$

---

**WRC-cas-rel**

$$\begin{array}{c|c|c} & [x] :=_{\texttt{rel}} 0; [y] :=_{\texttt{rel}} 0; & \\ [x] :=_{\texttt{rel}} 1; & a := \mathbf{cas}_{\texttt{rel,acq}}(y, 1, 2) & b :=_{\texttt{acq}} [y]; \\ [y] :=_{\texttt{rel}} 1 & & c :=_{\texttt{acq}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 1, b = 0, c = 0],$$

$$[a = 1, b = 0, c = 1], [a = 1, b = 0, c = 1],$$

$$[a = 1, b = 1, c = 1], [a = 1, b = 1, c = 1],$$

$$[a = 1, b = 2, c = 1].$$

---

**WRC-cas-rlx**

$$\begin{array}{c|c|c} & [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; & \\ [x] :=_{\texttt{rlx}} 1; & a := \mathbf{cas}_{\texttt{rlx,rlx}}(y, 1, 2) & b :=_{\texttt{acq}} [y]; \\ [y] :=_{\texttt{rel}} 1 & & c :=_{\texttt{rlx}} [x] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 1, b = 0, c = 0],$$

$$[a = 1, b = 0, c = 1], [a = 1, b = 0, c = 1],$$

$$[a = 1, b = 1, c = 1], [a = 1, b = 1, c = 1],$$

$$[a = 1, b = 2, c = 1].$$

## A.7 “Значения из воздуха”, OOTA

**OTA-lb**

$$\begin{array}{l||l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y]; & b :=_{\texttt{rlx}} [x] \\ [x] :=_{\texttt{rlx}} a; & [y] :=_{\texttt{rlx}} b \end{array}$$

C/C++11 MM разрешает для этой программы любой результат, в котором $a = b$.

---

**OTA-if**

$$\begin{array}{l||l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ a :=_{\texttt{rlx}} [y]; & b :=_{\texttt{rlx}} [x]; \\ \textbf{if } a & \textbf{if } b \\ \textbf{then } [x] :=_{\texttt{rlx}} 1 & \textbf{then } [y] :=_{\texttt{rlx}} 1 \\ \textbf{else } a := 0 & \textbf{else } b := 0 \\ \textbf{fi} & \textbf{fi} \end{array}$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0], [a = 1, b = 1].$$

## A.8 Независимые записи, WR

**WR-rlx**

$$\begin{array}{l||l} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \\ [y] :=_{\texttt{rlx}} 2 & [x] :=_{\texttt{rlx}} 2 \\ a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 1, b = 2], [a = 1, b = 2], [a = 1, b = 2], [a = 2, b = 2].$$

---

**WR-rlx-rel**

$$\begin{array}{c|c|c} [x] :=_{\texttt{rlx}} 0; & [y] :=_{\texttt{rlx}} 0; \\ [x] :=_{\texttt{rlx}} 1; & [y] :=_{\texttt{rlx}} 1; \\ [y] :=_{\texttt{rel}} 2 & [x] :=_{\texttt{rel}} 2 \\ a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 1, b = 2], [a = 1, b = 2], [a = 1, b = 2], [a = 2, b = 2].$$

---

**WR-rel**

$$\begin{array}{c|c|c} [x] :=_{\texttt{rel}} 0; & [y] :=_{\texttt{rel}} 0; \\ [x] :=_{\texttt{rel}} 1; & [y] :=_{\texttt{rel}} 1; \\ [y] :=_{\texttt{rel}} 2 & [x] :=_{\texttt{rel}} 2 \\ a :=_{\texttt{acq}} [x]; & b :=_{\texttt{acq}} [y] \end{array}$$

Результаты в C/C++11 MM:

$$[a = 1, b = 2], [a = 1, b = 2], [a = 1, b = 2], [a = 2, b = 2].$$

## A.9 Спекулятивное исполнение, SE

**SE-simple**

$$\begin{array}{l|l|l} [x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; [z] :=_{\texttt{rlx}} 0; & & \\ a :=_{\texttt{rlx}} [x]; & & b :=_{\texttt{rlx}} [y]; \\ \textbf{if } a & & \textbf{if } b \\ \textbf{then } [z] :=_{\texttt{rlx}} 1; & & \textbf{then } [x] :=_{\texttt{rlx}} 1 \\ \quad [y] :=_{\texttt{rlx}} 1 & & \textbf{else skip} \\ \textbf{else } [y] :=_{\texttt{rlx}} 1 & & \textbf{fi} \\ \textbf{fi} & & \end{array}$$

$$c :=_{\texttt{rlx}} [z]$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 0, b = 1, c = 0],$$

$$[a = 1, b = 1, c = 1].$$

**SE-prop**

$$
\begin{array}{l||l}
[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; [z] :=_{\texttt{rlx}} 0; & \\
a :=_{\texttt{rlx}} [x]; & b :=_{\texttt{rlx}} [y]; \\
\textbf{if } a & \textbf{if } b \\
\textbf{then } [z] :=_{\texttt{rlx}} 1; & \textbf{then } [x] :=_{\texttt{rlx}} 1 \\
\quad a :=_{\texttt{rlx}} [z]; & \textbf{else skip} \\
\quad [y] :=_{\texttt{rlx}} a & \textbf{fi} \\
\textbf{else } [y] :=_{\texttt{rlx}} 1 & \\
\textbf{fi} & \\
\multicolumn{2}{c}{c :=_{\texttt{rlx}} [z]}
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 0, b = 1, c = 0],$$

$$[a = 1, b = 1, c = 1].$$

**SE-nested**

$$
\begin{array}{l||l}
[x] :=_{\texttt{rlx}} 0; [y] :=_{\texttt{rlx}} 0; [z] :=_{\texttt{rlx}} 0; [f] :=_{\texttt{rlx}} 0; & \\
a :=_{\texttt{rlx}} [x]; & \\
\textbf{if } a & \\
\textbf{then } d :=_{\texttt{rlx}} [f]; & b :=_{\texttt{rlx}} [y]; \\
\quad \textbf{if } d & \textbf{if } a \\
\quad \textbf{then } [z] :=_{\texttt{rlx}} 1; & \textbf{then } [f] :=_{\texttt{rlx}} 1; \\
\quad\quad [y] :=_{\texttt{rlx}} 1 & \quad [x] :=_{\texttt{rlx}} 1 \\
\quad \textbf{else } [y] :=_{\texttt{rlx}} 1 & \textbf{else skip} \\
\quad \textbf{fi} & \textbf{fi} \\
\textbf{else } [y] :=_{\texttt{rlx}} 1 & \\
\textbf{fi} & \\
\multicolumn{2}{c}{c :=_{\texttt{rlx}} [z]}
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0, d = \bot], [a = 0, b = 1, c = 0, d = \bot],$$

$$[a = 1, b = 1, c = 0, d = 0], [a = 1, b = 1, c = 1, d = 1].$$

## A.10 ARM-weak

$$
\begin{array}{ccc}
[x] := 0; [y] := 0; & & \\
a := [x]; \ //1 & \| \ b := [x]; & \| \ c := [y]; \\
[x] := 1 & \| \ [y] := b & \| \ [x] := c
\end{array}
$$

Результаты в C/C++11 MM:

$$[a = 0, b = 0, c = 0], [a = 0, b = 1, c = 0], [a = 0, b = 1, c = 1], [a = 1, b = 1, c = 1].$$

## A.11 Блокировки

### Блокировка Деккера

$$
\begin{array}{lll}
[x] :=_{\texttt{rel}} 0; [y] :=_{\texttt{rel}} 0; [d] :=_{\texttt{na}} 0; & & \\
[x] :=_{\texttt{rel}} 1; & \| \ [y] :=_{\texttt{rel}} 1; & \\
a :=_{\texttt{acq}} [y]; & \| \ b :=_{\texttt{acq}} [x]; & \\
\textbf{if } a == 0 & \| \ \textbf{if } b == 0 & \\
\textbf{then } [d] :=_{\texttt{na}} 5 & \| \ \textbf{then } [d] :=_{\texttt{na}} 6 & \\
\textbf{else skip} & \| \ \textbf{else skip} & \\
\textbf{fi} & \| \ \textbf{fi} &
\end{array}
$$

Результаты в C/C++11 MM:

$$\textbf{undefined behavior}.$$

**Блокировка Коэна [66]**

$$[x] :=_{\texttt{rel}} 0; [y] :=_{\texttt{rel}} 0; [d] :=_{\texttt{na}} 0;$$

| | |
|---|---|
| $[x] :=_{\texttt{rel}}$ **choice** 1 2; | $[y] :=_{\texttt{rel}}$ **choice** 1 2; |
| **repeat** $[y]_{\texttt{acq}}$ **end**; | **repeat** $[x]_{\texttt{acq}}$ **end**; |
| $a :=_{\texttt{acq}} [x]$; | $c :=_{\texttt{acq}} [x]$; |
| $b :=_{\texttt{acq}} [y]$; | $d :=_{\texttt{acq}} [y]$; |
| **if** $a == b$ | **if** $c \neq d$ |
| **then** $[d] :=_{\texttt{na}} 5$ | **then** $[d] :=_{\texttt{na}} 6$ |
| **else skip** | **else skip** |
| **fi** | **fi** |

Результаты в C/C++11 MM:

$$[a = 1, b = 1, c = 1, d = 1], [a = 1, b = 2, c = 1, d = 2],$$

$$[a = 2, b = 1, c = 2, d = 1], [a = 2, b = 2, c = 2, d = 2].$$

# Приложение Б. Правила переходов и вспомогательные функции машины ARMv8 POP

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \bot \quad path.\mathsf{last} < size(cmds) \\ \exists path'.(tape(path') \neq \bot \vee path' = []) \wedge path \in \mathsf{next\text{-}path}(path', cmds, tape) \\ tape' \triangleq tape[path \mapsto \mathsf{get\text{-}new\text{-}tapecell}(cmds[path.\mathsf{last}])] \end{array}}{Prog[tid \mapsto cmds] \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Fetch instruction } tid\ path} \langle M_{\mathrm{POP}}, iordf, tapef[tid \mapsto tape']\rangle}$$

$$\frac{\begin{array}{c} e \in Evt \quad \neg Prop(tid,e) \quad \forall e' <_{Ord} e.\ Prop(tid,e') \quad Prop' \triangleq Prop \cup \{(tid, e)\} \\ Ord' \triangleq (Ord \cup \{(e, e') \mid Prop(tid,e') \wedge \neg Prop(e.tid,e'), e \not\leftrightarrow e', \neg(e' <_{Ord} e)\})^{+} \end{array}}{Prog \vdash \langle\langle Evt, Ord, Prop\rangle, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Propagate } e\ tid} \langle\langle Evt, Ord', Prop'\rangle, iordf, tapef\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{If\ none}\ k \quad val \triangleq [\![expr]\!]_{com} \in \mathbb{Z} \\ \mathsf{prev\text{-}branches\text{-}committed}(path, tape) \\ (st_{\mathrm{ifgoto}}, tape') \triangleq \mathsf{tape\text{-}upd\text{-}IfGoto}(val, k, path, tape) \\ M'_{\mathrm{POP}} \triangleq \mathsf{delete\text{-}upd\text{-}reads}(tid, tape', M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Branch commit } tid\ path} \langle M'_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tapef(tid, path) = \mathsf{F\ none}\ \mathtt{ld} \quad tapef' \triangleq tapef[(tid, path) \mapsto \mathsf{F\ com}\ \mathtt{ld}] \\ \mathsf{prev\text{-}reads\text{-}committed}(path, tapef(tid)) \quad \mathsf{prev\text{-}fences\text{-}committed}(path, tape(tid)) \\ \mathsf{prev\text{-}branches\text{-}committed}(path, tapef(tid)) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Fence commit } \mathtt{ld}\ tid\ path} \langle M_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tapef(tid, path) = \mathsf{F\ none}\ \mathtt{sy} \quad tapef' \triangleq tapef[(tid, path) \mapsto \mathsf{F\ com}\ \mathtt{sy}] \\ \mathsf{prev\text{-}instr\text{-}committed}(path, tapef(tid)) \\ M'_{\mathrm{POP}} \triangleq \mathsf{accept\text{-}request}(\langle tid, path, \mathsf{dmb}\rangle, M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Fence commit } \mathtt{sy}\ tid\ path} \langle M'_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tapef(tid, path) = \mathsf{W\ none} \quad Prog(tid)[path.\mathsf{last}] = \text{``}[expr_0] := expr_1\text{''} \\ [\![expr_0]\!] = \ell \quad [\![expr_1]\!] = val \quad tapef' = tapef[(tid, path) \mapsto \mathsf{W\ (pending}\ \ell\ val)] \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Write pending } tid\ path\ \ell\ val} \langle M_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{W\ (pending}\ \ell\ val) \quad cmds \triangleq Prog(tid) \\ cmds[path.\mathsf{last}] = \text{``}[expr_0] := expr_1\text{''} \quad [\![expr_0]\!]_{com} = \ell \quad [\![expr_1]\!]_{com} = val \\ \mathsf{prev\text{-}fences\text{-}committed}(path, tape) \quad \mathsf{prev\text{-}branches\text{-}committed}(path, tape) \\ \mathsf{prev\text{-}fully\text{-}determined}(path, tape) \quad \mathsf{no\text{-}prev\text{-}restartable\text{-}reads\text{-}from\text{-}loc}(\ell, path, tape) \\ im \triangleq \mathsf{no\text{-}following\text{-}com\text{-}writes\text{-}to\text{-}loc}(\ell, path, tape) \\ tape' \triangleq \mathsf{tape\text{-}upd\text{-}Wcom}(im, \ell, val, cmds, tid, path, tape) \\ tapef' = tapef[tid \mapsto tape'] \quad M''_{\mathrm{POP}} \triangleq \mathsf{delete\text{-}upd\text{-}reads}(tid, tape', M_{\mathrm{POP}}) \\ M'_{\mathrm{POP}} \triangleq \underline{\mathsf{if}}\ im\ \underline{\mathsf{then}}\ \mathsf{accept\text{-}request}(\langle tid, path, \mathsf{wr}\ \ell : val\rangle, M''_{\mathrm{POP}})\ \underline{\mathsf{else}}\ M''_{\mathrm{POP}} \end{array}}{\langle M_{\mathrm{POP}}, iordf, tapef, Prog\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Write commit } tid\ path\ \ell\ val} \langle M'_{\mathrm{POP}}, iordf, tapef', Prog\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{R\ none} \\ Prog(tid)[path.\mathsf{last}] = \text{``}reg = [expr]\text{''} \quad [\![expr]\!] = \ell \quad e \triangleq \langle tid, path, \mathsf{rd}\ \ell\rangle \\ tapef' = tapef[(tid, path) \mapsto \mathsf{R\ (requested}\ \ell)] \quad \mathsf{prev\text{-}fences\text{-}committed}(path, tape) \\ iord' \triangleq append(e, iordf(tid)) \quad iordf' = iordf[tid \mapsto iord'] \quad M'_{\mathrm{POP}} \triangleq \mathsf{accept\text{-}request}(e, M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Read issue}\ tid\ path\ \ell} \langle M'_{\mathrm{POP}}, iordf', tapef'\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{R\ (requested}\ \ell) \quad e \triangleq \langle tid, path, \mathsf{rd}\ \ell\rangle \quad e' \triangleq \langle tid', path', \mathsf{wr}\ \ell : val\rangle \\ \langle Evt, Ord, Prop\rangle \triangleq M_{\mathrm{POP}} \quad \{e, e'\} \subseteq Evt \quad e' <_{Ord} e \quad \mathsf{propagated\text{-}to\text{-}same\text{-}threads}(e, e', Prop) \\ \forall e^*, e' <_{Ord} e^* <_{Ord} e, get\ell(e^*) \neq \ell \wedge \mathsf{fully\text{-}propagated}(e^*, Prop) \\ \neg\mathsf{prev\text{-}read\text{-}from\text{-}other\text{-}write}(\ell, e', tid, path, iordf(tid)) \\ tape' \triangleq \mathsf{tape\text{-}upd\text{-}Rsat}(\mathsf{pln}, \ell, val, cmds, tid, path, tid', path', tape) \\ tapef' = tapef[tid \mapsto tape'] \quad M'_{\mathrm{POP}} \triangleq \mathsf{delete\text{-}upd\text{-}reads}(tid, tape', M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Read satisfy}\ tid\ path\ tid'\ path'\ \ell\ val} \langle M'_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{R\ (requested}\ \ell) \quad e \triangleq \langle tid, path, \mathsf{rd}\ \ell\rangle \quad e' \triangleq \langle tid', path', \mathsf{wr}\ \ell : val\rangle \\ \langle Evt, Ord, Prop\rangle \triangleq M_{\mathrm{POP}} \quad \{e, e'\} \subseteq Evt \quad e' <_{Ord} e \quad \mathsf{propagated\text{-}to\text{-}same\text{-}threads}(e, e', Prop) \\ \forall e^*, e' <_{Ord} e^* <_{Ord} e, get\ell(e^*) \neq \ell \wedge \mathsf{fully\text{-}propagated}(e^*, Prop) \\ \mathsf{prev\text{-}read\text{-}from\text{-}other\text{-}write}(\ell, e', tid, path, iordf(tid)) \\ tape' \triangleq tape[path \mapsto \mathsf{R\ none}] \quad tapef' = tapef[tid \mapsto tape'] \\ M'_{\mathrm{POP}} \triangleq \mathsf{delete\text{-}upd\text{-}reads}(tid, tape', M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Read satisfy (fail)}\ tid\ path\ tid'\ path'\ \ell\ val} \langle M'_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{R\ none} \quad tape(path') = \mathsf{W\ (pending}\ \ell\ val) \\ path' < path \quad cmds \triangleq Prog(tid) \quad cmds[path.\mathsf{last}] = \text{``}reg = [expr]\text{''} \quad [\![expr]\!]^{path} = \ell \\ \mathsf{no\text{-}writes\text{-}to\text{-}loc\text{-}in\text{-}between}(\ell, cmds, path', path, tape) \\ \mathsf{no\text{-}different\text{-}write\text{-}reads\text{-}in\text{-}between}(\ell, tid, path', path, tape) \\ tape' \triangleq \mathsf{tape\text{-}upd\text{-}Rsat}(\mathsf{inflight}, \ell, val, cmds, tid, path, tid, path', tape) \\ tapef' = tapef[tid \mapsto tape'] \quad M_{\mathrm{POP}} \triangleq \mathsf{delete\text{-}upd\text{-}reads}(tid, tape', M_{\mathrm{POP}}) \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Read satisfy from in-flight write}\ tid\ path\ path'\ \ell\ val} \langle M'_{\mathrm{POP}}, iordf, tapef'\rangle}$$

$$\frac{\begin{array}{c} tape \triangleq tapef(tid) \quad tape(path) = \mathsf{R\ (sat}\ sat\text{-}state\ \langle tid', path', \mathsf{wr}\ \ell : val\rangle) \\ sat\text{-}state \neq \mathsf{com} \quad cmds \triangleq Prog(tid) \quad \mathsf{prev\text{-}fully\text{-}determined}(path, tape) \\ \mathsf{prev\text{-}fences\text{-}committed}(path, tape) \quad \mathsf{prev\text{-}branches\text{-}committed}(path, tape) \\ path'' = \mathsf{max}\{path^* < path | cmds[path^*.\mathsf{last}] = \text{``}[expr_0] := expr_1\ [\![expr_0]\!]^{path^*}_{com} = \ell\} \\ \underline{\text{if}}\ (tid', path') = (tid, path'')\ \underline{\text{then}}\ tape(path')\ \text{is fully determined}\ \underline{\text{else}}\ tape(path')\ \text{is committed} \\ \mathsf{reads\text{-}in\text{-}between\text{-}committed}(path'', path, tape, cmds) \\ tape' \triangleq tape[path \mapsto \mathsf{R\ (sat\ com}\ \langle tid', path', \mathsf{wr}\ \ell : val\rangle)] \quad tapef' = tapef[tid \mapsto tape'] \end{array}}{Prog \vdash \langle M_{\mathrm{POP}}, iordf, tapef\rangle \xrightarrow[\mathrm{ARM}]{\textbf{Read commit}\ tid\ path} \langle M_{\mathrm{POP}}, iordf, tapef'\rangle}$$

prev-instr-committed$(path, tape) \triangleq \forall path' < path, tape(path')$ is committed.
prev-reads-committed$(path, tape) \triangleq \forall path' < path, tape(path') = \mathsf{R}\ _ \Rightarrow tape(path')$ is committed.
prev-fences-committed$(path, tape) \triangleq \forall path' < path, tape(path') = \mathsf{F}\ st_{\text{fence}}\ _ \Rightarrow st_{\text{fence}} = \mathsf{com}$.
prev-branches-committed$(path, tape) \triangleq \forall path' < path, tape(path') = \mathsf{If}\ st_{\text{ifgoto}}\ _ \Rightarrow st_{\text{ifgoto}} \neq \mathsf{none}$.
prev-fully-determined$(path, tape) \triangleq \forall path' < path, tape(path')$ has a fully determined address.
no-prev-restartable-reads-from-loc$(\ell, path, tape) \triangleq$
$\forall path' < path, reg, expr, tape(path') = \mathsf{R}\ st_{\text{read}}, cmds[path'.\mathsf{last}] =$ ‘‘$reg = [expr]$ $[\![expr]\!] = \ell \Rightarrow$
$st_{\text{read}} = \mathsf{sat}\ \_\ \_ \wedge tape(path')$ can’t be restarted.

no-following-com-writes-to-loc$(\ell, path, tape) \triangleq \nexists path' > path,\ tape(path') = \mathsf{W}\ (\mathsf{com}\ _\ \ell\ _)$.

tape-upd-IfGoto$(val, k, path, tape) \triangleq$
let $st_{\text{ifgoto}} \triangleq$ if $val \neq 0$ then taken else ignored in
let $path_{drop} \triangleq$ if $val \neq 0$ then append$(path.\mathsf{last} + 1, path)$ else append$(path.\mathsf{last} + k, path)$ in
$(st_{\text{ifgoto}}, \lambda\ path' \rightarrow$
if prefix$(path_{drop}, path')$ then $\bot$
elif $path' = path$ then If $st_{\text{ifgoto}}\ k$
else $tape(path')$).

reads-in-between-committed$(path'', path, tape, cmds) \triangleq$
$\forall path^* > path'', path^* < path,$
$cmds[path^*.\mathsf{last}] =$ ‘‘$reg = [expr]$ $[\![expr]\!]^{path^*}_{com} = x \Rightarrow tape(path^*)$ is committed.

no-writes-to-loc-in-between$(\ell, cmds, path', path, tape) \triangleq$
$\nexists path^* > path', path^* < path, cmds[path^*.\mathsf{last}] =$ ‘‘$[expr_0] := expr_1$ $[\![expr_0]\!]^{path^*} = \ell$.

no-different-write-reads-in-between$(\ell, tid', path', path, tape) \triangleq$
$\nexists tid'' \neq tid', path'' \neq path', path^* > path', path^* < path, tape(path^*) = \mathsf{R}\ (\mathsf{sat}\ \_\ \langle tid'', path'', \mathsf{wr}\ \ell : \_\rangle)$.

delete-upd-reads$(tid, tape, \langle Evt, Ord, Prop\rangle) \triangleq$
let *to-delete* $\triangleq \{e \in Evt \mid e.tid = tid, tape'(e.path) \neq \mathsf{R}\ (\mathsf{requested}\ \_)\}$ in
let $Evt' \triangleq Evt \setminus$ *to-delete* in
let $Prop' = Prop \setminus (\mathbb{N} \times$ *to-delete*$)$ in
let $Ord' = (Ord \setminus (Evt \times$ *to-delete* $\cup$ *to-delete* $\times Evt) \setminus \hookrightarrow)^+$ in
$\langle Evt', Ord', Prop'\rangle$.

accept-request$(e, \langle Evt, Ord, Prop\rangle) \triangleq$
let $Evt' = Evt \cup \{e\}$ in
let $Prop' = Prop[tid \mapsto Prop(tid) \cup \{e\}]$ in
let $Ord' = (Ord \cup \{(e', e) | e' \in Prop(tid), e' \not\hookrightarrow e\})^+$ in
$\langle Evt', Ord', Prop'\rangle$.

propagated-to-same-threads$(e, e', Prop) \triangleq \{tid \mid Prop(tid, e)\} = \{tid \mid Prop(tid, e')\}$
fully-propagated$(e, Prop) \triangleq \forall tid, Prop(tid, e) \vee \nexists e'.Prop(tid, e')$.
$get\ell(e) \triangleq$ match $e$ with $\langle \_, \_, \mathsf{rd}\ \ell\rangle \mid \langle \_, \_, \mathsf{wr}\ \ell : \_\rangle \to \ell \mid \_ \to \bot$ end.

prev-read-from-other-write$(\ell, e', tid, path, iord) \triangleq$
$\exists path^* < path,$
$last\_index(\langle tid, path^*, \mathsf{rd}\ \ell\rangle, iord) > last\_index(\langle tid, path, \mathsf{rd}\ \ell\rangle, iord),$
$tape(path^*) = \mathsf{R}\ (\mathsf{sat}\ \_\ req), req \neq e').$

get-new-tapecell$(S) \triangleq$
match $S$ with
|"*reg* := [*expr*]" $\to$ R none
|"[$expr_0$] := $expr_1$" $\to$ W none
|"fence($fmod_{\text{ARM}}$)" $\to$ F none $fmod_{\text{ARM}}$
|"if *expr* goto *k*" $\to$ If none *k*
|"*reg* = *expr*" $\to$ Assign
|"nop" $\to$ Nop
end.

$e <_{Ord} e' \triangleq (e, e') \in Ord.$
$\langle tid, path, reqinfo\rangle \hookrightarrow \langle tid', path', reqinfo'\rangle \triangleq$
if $\mathtt{sy} \in \{reqinfo, reqinfo'\}$ then $false$
elif $reqinfo.\ell = reqinfo'.\ell \neq \bot$ then $false$
else $true$.

next-path$(path, cmds, tape) \triangleq filter(\lambda path' \to path'.\mathsf{last} < size(cmds),$
if $path = []$ then $\{[0]\}$
elif $tape(path) = \bot$ then $\varnothing$
elif $\exists k, tape(path) = \mathsf{If\ none}\ k$ then $\{snoc(path, path.\mathsf{last} + 1), snoc(path, path.\mathsf{last} + k)\}$
elif $\exists k, tape(path) = \mathsf{If\ taken}\ k$ then $\{snoc(path, path.\mathsf{last} + k)\}$
else $\{snoc(path, path.\mathsf{last} + 1)\}$).

tape-upd-restart$(cmds, tid, tape) \triangleq$
$fixpoint(\lambda tape' \to$
$\lambda path \to$
if $cmds[path.\mathsf{last}]$ = "*reg* = [*expr*]" $\wedge$ $[\![expr]\!]^{path} = \bot$
then R none
elif $\exists path'', tape'(path) = \mathsf{R}\ (\mathsf{sat\ inflight}\ \langle tid, path'', \mathsf{wr}\ \_ : \_\rangle) \wedge tape'(path'') = \mathsf{W\ none}$
then R none
elif $cmds[path.\mathsf{last}]$ = "[$expr_0$] = $expr_1$" $\wedge$ ($[\![expr_0]\!]^{path} = \bot \vee [\![expr_1]\!]^{path} = \bot$)
then W none
else $tape'(path))(tape)$.

tape-upd-Wcom$(im, \ell, val, cmds, tid, path, tape) \triangleq$
  tape-upd-restart$(cmds, tid,$
    $\lambda path' \rightarrow$
      $\underline{\text{if}}\ path' = path\ \underline{\text{then}}$ W (com $im\ \ell\ val$)
      $\underline{\text{elif}}\ path' < path\ \underline{\text{then}}\ tape(path')$
      $\underline{\text{elif}}\ tape(path') =$ R (sat inflight $\langle tid, path,$ wr $\ell : val\rangle$)
        $\underline{\text{then}}\ tape(path') =$ R (sat pln $\langle tid, path,$ wr $\ell : val\rangle$)
      $\underline{\text{elif}}\ tape(path') =$ R (requested $\ell$) $\underline{\text{then}}$ R none
      $\underline{\text{elif}}\ \exists path'' < path, tape(path') =$ R (sat inflight $\langle tid, path'',$ wr $\ell : \_\rangle$) $\underline{\text{then}}$ R none
      $\underline{\text{elif}}\ \exists tid'', path'', \neg(tid'' = tid \wedge path'' \geqslant path) \wedge tape(path') =$ R (sat pln $\langle tid'', path'',$ wr $\ell : \_\rangle$)
        $\underline{\text{then}}$ R none
      $\underline{\text{else}}\ tape(path'))$.

tape-upd-Rsat$(sat\text{-}state, \ell, val, cmds, tid, path, tid', path', tape) \triangleq$
  tape-upd-restart$(cmds, tid,$
    $\lambda path'' \rightarrow$
      $\underline{\text{if}}\ path'' = path\ \underline{\text{then}}$ R (sat $sat\text{-}state\ \langle tid', path',$ wr $\ell : val\rangle$)
      $\underline{\text{elif}}\ path'' < path\ \underline{\text{then}}\ tape(path'')$
      $\underline{\text{elif}}\ \exists path^* < path, tape(path'') =$ R (sat inflight $\langle tid, path^*,$ wr $\ell : \_\rangle$) $\underline{\text{then}}$ R none
      $\underline{\text{elif}}\ \exists tid^*, path^*, \neg(tid^* = tid \wedge path^* > path) \wedge \neg(tid^* = tid' \wedge path^* = path') \wedge$
        $tape(path'') =$ R (sat pln $\langle tid^*, path^*,$ wr $\ell : \_\rangle$) $\underline{\text{then}}$ R none
      $\underline{\text{else}}\ tape(path''))$.

# Приложение B. Доказательство вспомогательных лемм о симуляции модели ARM+τ

## B.1 Базовые леммы

**Лемма 5.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{pre}}.\ \exists \mathbf{p}'.\mathit{Prog} \vdash \mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}' \wedge (\mathbf{a}, \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}$.

*Доказательство.* Зафиксируем $\mathbf{a}, \mathbf{p}$. Поскольку $(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{pre}}$, то $\mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a}, \mathbf{p})$ выполняется. Как следствие, существует единственный поток с идентификатором *tid*, плёнка $\mathit{tape} = \mathbf{a}.\mathit{tapef}(\mathit{tid})$ и путь $\mathit{path} = \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{path}$ такие, что экземпляр $\mathit{tape}(\mathit{path})$ завершён. Далее в доказательстве мы конструируем состояние обещающей машины $\mathbf{p}'$ такое, что оно удовлетворяет утверждению теоремы.

Введём следующие обозначения:

$$
\begin{array}{lcl}
\mathit{cmds} & \triangleq & \mathit{Prog}(\mathit{tid}); \\
\langle \mathit{view}_{\text{cur}}, \mathit{view}_{\text{acq}}, \mathit{view}_{\text{rel}} \rangle & \triangleq & \mathbf{p}.\mathcal{TS}(\mathit{tid}).V; \\
\mathit{view}'_{\text{cur}}, \mathit{view}'_{\text{acq}}, \mathit{view}'_{\text{rel}} & \triangleq & \mathbf{p}'.\mathcal{TS}(\mathit{tid}).V; \\
\mathit{path}' & \triangleq & \mathbf{p}'.\mathcal{TS}(\mathit{tid}).\mathit{path}; \\
\mathit{st} & \triangleq & \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{st}; \\
\mathit{st}' & \triangleq & \mathbf{p}'.\mathcal{TS}(\mathit{tid}).\mathit{st}; \\
\mathit{promises}' & \triangleq & \mathbf{p}'.\mathcal{TS}(\mathit{tid}).\mathit{promises}; \\
\mathit{promises} & \triangleq & \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{promises}; \\
H & \triangleq & \mathbf{a}.H.
\end{array}
$$

Мы покажем, что обещающая машина в состоянии $\mathbf{p}$ может сделать переход, связанный с $\mathit{tape}(\mathit{path})$, в состояние $\mathbf{p}'$, и при этом будет выполняться $(\mathbf{a}, \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}$. Далее мы проведём разбор вариантов состояния экземпляра $\mathit{tape}(\mathit{path})$.

Очевидно, что для любого нового состояния $\mathbf{p}'$, такого что $\mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}'$, выполняется $(\mathbf{a}, \mathbf{p}') \in \mathcal{I}_{\text{reach}} \cap \mathcal{I}_{\textbf{mem3}}$. Выполнение $\mathcal{I}_{\text{prefix}}(\mathbf{a}, \mathbf{p}')$ следует из того, что $\mathit{path}' \in \mathsf{next\text{-}path}(\mathit{path}, \mathit{cmds}, \mathit{tape})$, $\mathcal{I}^{\text{ARM}}_{\text{Next-Committed}}(\mathbf{a})$, и исполнение экземпляра $\mathit{tape}(\mathit{path})$ завершено. Поскольку переход $\mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}'$ будет использован для того, чтобы "нагнать" исполнение машины ARM+τ, то он не является переходом **Завершение записи**. Как следствие, $\mathbf{p}'.M = \mathbf{p}.M$, и утверждение $\mathcal{I}_{\textbf{mem2}}(\mathbf{a}, \mathbf{p}')$ следует из $\mathcal{I}_{\textbf{mem2}}(\mathbf{a}, \mathbf{p})$. Кроме того, очевидно, что будет выполняться либо $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}, \mathbf{p}')$, либо $\mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a}, \mathbf{p}')$ в зависимости от завершённости экземпляра $\mathit{tape}(\mathit{path}')$.

Рассмотрим варианты состояния (завершённого) экземпляра $tape(path)$.

– $tape(path) = \mathsf{Nop}$, $tape(path) = \mathsf{Assign}$ или $tape(path) = \mathsf{If}\ st_{\text{ifgoto}}\ k$.
Во всех трёх случаях переход обещающей машины $\mathbf{p} \xrightarrow[\text{Promise}]{} \mathbf{p}'$ будет внутренним, т.е. $\varepsilon$-переходом. Поскольку $\varepsilon$-переход не меняет состояние памяти, то утверждение $\mathcal{I}_{\mathbf{mem1}}(\mathbf{a}, \mathbf{p}')$ непосредственно следует из $\mathcal{I}_{\mathbf{mem1}}(\mathbf{a}, \mathbf{p})$. Аналогично выполняются $\mathcal{I}_{\mathbf{view}}(\mathbf{a}, \mathbf{p}')$ (поскольку $\mathbf{p}'.\mathcal{TS}(tid).V = \mathbf{p}.\mathcal{TS}(tid).V$), $\mathcal{I}_{\mathbf{state}}(\mathbf{a}, \mathbf{p}')$ (поскольку $\mathbf{p}'.\mathcal{TS}(tid).st = \mathbf{p}.\mathcal{TS}(tid).st$ и $\mathsf{regf}_{\mathsf{com}}(cmds(tid), tape, path') = \mathsf{regf}_{\mathsf{com}}(cmds(tid), tape, path')$) и $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}')$ (поскольку экземпляр $tape(path)$ не является ни записью, ни барьером).

– $tape(path) = \mathsf{F\ com}\ \texttt{ld}$.
Обещающая машина делает переход, соответствующий приобретающему барьеру памяти. Как следствие, новый базовый фронт $view'_{\text{cur}}$ равен старому приобретающему фронту $view_{\text{acq}}$. Утверждения $\mathcal{I}_{\mathbf{mem1}}(\mathbf{a}, \mathbf{p}')$ и $\mathcal{I}_{\mathbf{state}}(\mathbf{a}, \mathbf{p}')$ выполняются, поскольку переход не меняет состояния локальных переменных и памяти.
Проверим, что выполняется $\mathcal{I}_{\mathbf{view}}(\mathbf{a}, \mathbf{p}')$. Для этого нам нужно показать, что следующее утверждение верно:

$$
\begin{aligned}
&\forall tid', tape' = \mathbf{a}.tapef(tid'), path'' = \mathbf{p}.\mathcal{TS}(tid').path,\\
&\quad(\mathbf{p}'.\mathcal{TS}(tid').view_{\text{acq}} \leqslant \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(path'', tape', H)\\
&\qquad\sqcup \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid', path'', tape', H)) \wedge\\
&\quad(\mathbf{p}'.\mathcal{TS}(tid').view_{\text{cur}} \leqslant \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(\mathsf{last}\texttt{ld}(tape', path''), tape', H)\\
&\qquad\sqcup \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid', path'', tape', H)) \wedge\\
&\quad(\mathbf{p}'.\mathcal{TS}(tid').view_{\text{rel}} \leqslant \bigsqcup \mathsf{sat\text{-}reads\text{-}view}(\mathsf{last}\texttt{ldsy}(tape', path''), tape', H)\\
&\qquad\sqcup \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid', \mathsf{last}\texttt{sy}(tape', path''), tape', H))).
\end{aligned}
$$

Зафиксируем $tid', tape', path''$. Если $tid' \neq tid$, то утверждение следует из $\mathcal{I}_{\mathbf{view}}(\mathbf{a}, \mathbf{p})$ и $\mathbf{p}'.\mathcal{TS}(tid').V = \mathbf{p}.\mathcal{TS}(tid').V$. Пусть $tid' = tid$. Как следствие, $tape' = tape$ и $path'' = path'$. Также верно, что

- $\mathbf{p}'.\mathcal{TS}(tid).view_{\text{acq}} = \mathbf{p}.\mathcal{TS}(tid).view_{\text{acq}} = view_{\text{acq}}$,
- $\mathbf{p}'.\mathcal{TS}(tid).view_{\text{cur}} = \mathbf{p}.\mathcal{TS}(tid).view_{\text{acq}} = view_{\text{acq}}$,
- $\mathbf{p}'.\mathcal{TS}(tid).view_{\text{rel}} = \mathbf{p}.\mathcal{TS}(tid).view_{\text{rel}} = view_{\text{rel}}$.

Тогда утверждение в упрощённой форме выглядит так:

$$
\begin{aligned}
&(\mathit{view}_{\text{acq}} \leqslant \bigsqcup \textsf{sat-reads-view}(\mathit{path}', \mathit{tape}, H) \\
&\quad \sqcup \bigsqcup \textsf{com-writes-time}(\mathit{tid}, \mathit{path}', \mathit{tape}, H)) \wedge \\
&(\mathit{view}_{\text{acq}} \leqslant \bigsqcup \textsf{sat-reads-view}(\textsf{last}\texttt{ld}(\mathit{tape}, \mathit{path}'), \mathit{tape}, H) \\
&\quad \sqcup \bigsqcup \textsf{com-writes-time}(\mathit{tid}, \mathit{path}', \mathit{tape}, H)) \wedge \\
&(\mathit{view}_{\text{rel}} \leqslant \bigsqcup \textsf{sat-reads-view}(\textsf{last}\texttt{ldsy}(\mathit{tape}, \mathit{path}'), \mathit{tape}, H) \\
&\quad \sqcup \bigsqcup \textsf{com-writes-time}(\mathit{tid}, \textsf{last}\texttt{sy}(\mathit{tape}, \mathit{path}'), \mathit{tape}', H))).
\end{aligned}
$$

Первый и третий конъюнкты следует из того, что выполняется $\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p})$ и $\mathit{path}' > \mathit{path}$. Второй конъюнкт следует из определения $\textsf{last}\texttt{ld}(\mathit{tape}, \mathit{path}')$ и $\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p})$.

Утверждение $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}')$ выполняется, поскольку экземпляр $\mathit{tape}(\mathit{path})$ не является ни записью, ни `sy`-барьером.

– $\mathit{tape}(\mathit{path}) = \textsf{F com}\ \texttt{sy}$.

Обещающая машина делает переход, соответствующий высвобождающему барьеру памяти. Как следствие, новый высвобождающий фронт $\mathit{view}'_{\text{rel}}$ равен старому базовому фронту $\mathit{view}_{\text{cur}}$. Утверждения $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p}')$ и $\mathcal{I}_{\textbf{state}}(\mathbf{a}, \mathbf{p}')$ выполняются, поскольку переход не меняет состояния локальных переменных и памяти. То, что утверждение $\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p}')$ выполняется, может быть показано аналогичными выкладками, что и в случае $\mathit{tape}(\mathit{path}) = \textsf{F com}\ \texttt{ld}$.

Из верности утверждения $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p})$ следует, что не существует незавершённого экземпляра записи с путём $\mathit{path}^{write} > \mathit{path}$. Это означает, что выполняется $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}')$.

– $\mathit{tape}(\mathit{path}) = \textsf{R}\ (\textsf{sat com}\ \langle \mathit{tid}'' : \mathit{path}''@x, \mathit{val} \rangle)$.

Обещающая машина делает переход, соответствующий расслабленному чтению. Утверждения $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p}')$ и $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}')$ выполняются по тем же соображениям, что и в первом рассмотренном случае.

Поскольку все экземпляры с путями, меньше чем $\mathit{path}$, завершены, то экземпляр записи $(\mathit{tid}'', \mathit{path}'')$, из которого читает экземпляр $\mathit{tape}(\mathit{path})$, точно завершён. Введём следующие обозначения:

$$
\begin{aligned}
&\mathbf{a}.\mathit{tapef}(\mathit{tid}'', \mathit{path}'') = \textsf{W}\ (\textsf{com}\ \_\ x\ \mathit{val}); \\
&(\tau, \_, \mathit{view}') \triangleq \mathbf{a}.H(\mathit{tid}'', \mathit{path}'') \neq (\bot, \_, \bot).
\end{aligned}
$$

Наличие соответствующего сообщения $\langle x : \mathit{val}@\tau, \mathit{view} \rangle$ в памяти обещающей машины следует из $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p})$. То, что метка времени сообщения

не меньше, чем значение базового фронта потока, т.е. $\tau \geqslant \mathit{view}_{\text{cur}}(x)$, следует из $\mathcal{I}^{\text{ARM}}_{\text{View}}(\mathbf{a})$.

По определению переходов обещающей машины мы знаем, что $\mathit{view}'_{\text{cur}} = \mathit{view}_{\text{cur}} \sqcup [x@\tau]$, $\mathit{view}'_{\text{acq}} = \mathit{view}_{\text{acq}} \sqcup \mathit{view}, \mathit{view} \leqslant \mathit{view}'$ и $\mathit{st}' = \mathit{st}[\mathit{reg} \mapsto \mathit{val}]$. Утверждение $\mathcal{I}_{\textbf{state}}(\mathbf{a}, \mathbf{p}')$ выполняется, поскольку $\mathit{st}' = \mathit{st}[\mathit{reg} \mapsto \mathit{val}] = \mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path})[\mathit{reg} \mapsto \mathit{val}] = \mathsf{regf}_{\mathsf{com}}(\mathit{cmds}, \mathit{tape}, \mathit{path}')$.

Выполнение утверждения $\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p}')$ следует из определения функций com-writes-time и sat-reads-view.

– $\mathit{tape}(\mathit{path}) = \mathsf{W}\ (\mathsf{com}\ \mathit{im}\ x\ \mathit{val})$.

Обещающая машина делает переход, соответствующий выполнению ранее данного обещания. Мы знаем, что $(\tau, \_, \mathit{view}') \triangleq \mathbf{a}.H_\tau(\mathit{tid}, \mathit{path}) \neq \bot$, поскольку экземпляр записи завершён. То, что существует нужное обещание, т.е. $\langle x : \mathit{val}@\tau, \mathit{view}\rangle \in \mathbf{p}.\mathcal{TS}(\mathit{tid}).\mathit{promises}$, напрямую следует из $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p})$. Фронт $\mathit{view}$ равен $\mathit{view}_{\text{rel}} \sqcup [x@\tau]$, поскольку обещающая машина не имеет возможности делать обещания через высвобождающие барьеры памяти. Утверждение $\mathcal{I}_{\textbf{state}}(\mathbf{a}, \mathbf{p}')$ выполняется, поскольку верно $\mathcal{I}_{\textbf{state}}(\mathbf{a}, \mathbf{p}')$ и выполнение обещания не меняет состояния переменных. То, что $\mathit{view}_{\text{cur}}(x) < \tau$ следует из $\mathcal{I}^{\text{ARM}}_{\text{View}}(\mathbf{a})$. То, что $\mathit{view} \leqslant R'$ следует из того, как машина ARM+$\tau$ конструирует компоненту $H_{\mathsf{view}}$.

Мы знаем, что

$$\begin{aligned}
&\mathit{view}'_{\text{cur}} = \mathit{view}_{\text{cur}} \sqcup [x@\tau];\\
&\mathit{view}'_{\text{acq}} = \mathit{view}_{\text{acq}} \sqcup [x@\tau];\\
&\mathit{promises}' = \mathit{promises} \setminus \{\langle x : \mathit{val}@\tau, \mathit{view}\rangle\}.
\end{aligned}$$

Утверждение $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p}')$ следует из $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p})$ и $\mathit{tape}(\mathit{path}) = \mathsf{W}\ (\mathsf{com}\ \mathit{im}\ x\ \mathit{val})$. Утверждение $\mathcal{I}_{\textbf{view}}(\mathbf{a}, \mathbf{p}')$ следует из определений функций com-writes-time и sat-reads-view. Утверждение $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p}')$ верно, т.к. все экземпляры инструкций, чей путь меньше $\mathit{path}$, являются завершёнными.

То, что переход $\mathbf{p} \xrightarrow[\mathsf{Promise}]{} \mathbf{p}'$ сертифицируем, т.е. существует конечное число переходов потока $\mathit{tid}$, после которых все обещания потока выполнены, доказывается в приложении B.2. □

**Лемма 7.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}$.
$(\forall \mathbf{a}'.\ \mathit{Prog} \vdash \mathbf{a} \xrightarrow[ARM{+}\tau]{\neg\ \textbf{Write commit}} \mathbf{a}' \Rightarrow (\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}) \wedge$
$(\forall \mathbf{a}'.\ \mathit{Prog} \vdash \mathbf{a} \xrightarrow[ARM{+}\tau]{\textbf{Write commit}} \mathbf{a}' \Rightarrow \exists \mathbf{p}'.\ \mathit{Prog} \vdash \mathbf{p} \xrightarrow[\text{Promise}]{\textbf{Promise write}} \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I})$.

*Доказательство.* Рассмотрим первый конъюнкт. Зафиксируем $\mathbf{a}$, $\mathbf{p}$ и $\mathbf{a}'$ такие, что $\mathbf{a} \xrightarrow[ARM{+}\tau]{\neg\ \textbf{Write commit}\ \mathit{tid}} \mathbf{a}'$. Введём следующие обозначения:

$$\langle \mathit{path}', \mathit{st}, \langle \mathit{view}_{\text{cur}}, \mathit{view}_{\text{acq}}, \mathit{view}_{\text{rel}} \rangle, \mathit{promises} \rangle \triangleq \mathbf{p}.\mathcal{TS}(\mathit{tid});$$
$$\mathit{tape} \triangleq \mathbf{a}.\mathit{tapef}(\mathit{tid});$$

Нам нужно показать, что $(\mathbf{a}', \mathbf{p}) \in \mathcal{I} \cup \mathcal{I}_{\text{pre}}$. Из определений $\mathcal{I}_{\text{reach}}$ и $\mathcal{I}_{\textbf{mem3}}$ следует, что $(\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\text{reach}} \cap \mathcal{I}_{\textbf{mem3}}$ выполняется. Утверждения $\mathcal{I}_{\text{prefix}}(\mathbf{a}', \mathbf{p})$ и $\mathcal{I}_{\textbf{view}}(\mathbf{a}', \mathbf{p})$ выполняется, потому что переход $\mathbf{a} \xrightarrow[ARM{+}\tau]{} \mathbf{a}'$ не меняет завершённый префикс $\mathbf{a}.\mathit{tapef}(\mathit{tid})$. Утверждение $\mathcal{I}_{\text{com-SY}}(\mathbf{a}', \mathbf{p})$ следует из $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p})$ и требований правила **Завершение** `sy`**-барьера**.

Нам нужно показать, что выполняется либо $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}', \mathbf{p})$, либо $\mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a}', \mathbf{p})$. Если переход $\mathbf{a} \xrightarrow[ARM{+}\tau]{} \mathbf{a}'$ является **Уведомление потока** $\mathit{tid}''$ **о запросе** $e''$, для плёнка любого потока не меняется, а значит как следствие того, что $(\mathbf{a}, \mathbf{p}) \in \mathcal{I}$, выполняется $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}', \mathbf{p})$. Иначе, существует некоторый экземпляр $(\mathit{tid}, \mathit{path})$, часть исполнения которого выполняется на переходе $\mathbf{a} \xrightarrow[ARM{+}\tau]{} \mathbf{a}'$. Если $\mathit{path}' \neq \mathit{path}$, то очевидно выполняется $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}', \mathbf{p})$. Рассмотрим вариант $\mathit{path}' = \mathit{path}$. В этом случае выполняется либо $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}', \mathbf{p})$, либо $\mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a}', \mathbf{p})$, в зависимости от завершённости экземпляра $\mathbf{a}'.\mathit{tapef}(\mathit{tid}, \mathit{path})$.

Утверждение $(\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\textbf{mem1}} \cap \mathcal{I}_{\textbf{mem2}}$ верно, поскольку $\mathbf{a} \xrightarrow[ARM{+}\tau]{} \mathbf{a}'$ не является переходом **Завершение записи**, а значит набор сообщений записи в подсистеме памяти машины ARM+τ не меняется.

Так доказывается первый конъюнкт леммы.

Рассмотрим второй конъюнкт. Зафиксируем $\mathbf{a}$, $\mathbf{p}$ и $\mathbf{a}'$ такие, что $\mathbf{a} \xrightarrow[ARM{+}\tau]{\textbf{Write commit } tid\ path\ x\ val\ \tau} \mathbf{a}'$. Введём следующие обозначения:

$$
\begin{array}{lcl}
\langle path', st, \langle view_{\text{cur}}, view_{\text{acq}}, view_{\text{rel}}\rangle, promises\rangle & \triangleq & \mathbf{p}.\mathcal{TS}(tid);\\
view' & \triangleq & view_{\text{rel}} \sqcup [x@\tau];\\
promises' & \triangleq & promises \cup \{\langle x : val@\tau, view'\rangle\};\\
\mathcal{TS}' & \triangleq & \mathbf{p}.\mathcal{TS}[tid \mapsto \langle path', st, \langle view_{\text{cur}}, view_{\text{acq}}, view_{\text{rel}}\rangle, promises'\rangle];\\
tape & \triangleq & \mathbf{a}.tapef(tid);\\
H & \triangleq & \mathbf{a}.H;
\end{array}
$$

Мы знаем, что метка времени $\tau$ не была использована для сообщений к локации $x$, поскольку она использована машиной ARM+$\tau$ на этом шаге. В обещающей машине переход **Обещание записи** не имеет ограничений, (с точностью до сертификации, возможность которой доказывается в приложении B.2), поэтому возможен переход $\mathbf{p} \xrightarrow[\text{Promise}]{\textbf{Promise write } tid\ \langle x{:}val@\tau{,}view'\rangle} \mathbf{p}'$, где $\mathbf{p}' = \langle \mathbf{p}.M \cup \{\langle x : val@\tau, view'\rangle\}, \mathcal{TS}'\rangle$.

Нам нужно проверить, что выполняется $(\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}$. Как следствие $(\mathbf{a}, \mathbf{p}) \in \mathcal{I}$ и определений $\mathbf{a}'$ и $\mathbf{p}'$, выполняется $(\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{reach}} \cap \mathcal{I}_{\textbf{mem3}}$. Утверждения $\mathcal{I}_{\text{prefix}}(\mathbf{a}', \mathbf{p}'), \mathcal{I}_{\textbf{view}}(\mathbf{a}', \mathbf{p}'), \mathcal{I}_{\textbf{state}}(\mathbf{a}', \mathbf{p}')$ и $\mathcal{I}_{\text{com-SY}}(\mathbf{a}', \mathbf{p})$ выполняются, поскольку переход **Завершение записи** машины ARM+$\tau$ не меняет завершённый префикс $\mathbf{a}.tapef(tid)$. Утверждение $\mathcal{I}_{\text{Promise is up to ARM}}(\mathbf{a}', \mathbf{p}) \cup \mathcal{I}_{\text{Promise isn't up to ARM}}(\mathbf{a}', \mathbf{p})$ выполняется по тем же соображениям, что и в предыдущем рассмотренном варианте.

Утверждение $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}', \mathbf{p}')$ верно, поскольку верно $\mathcal{I}_{\textbf{mem1}}(\mathbf{a}, \mathbf{p})$, а изменения памяти машин отражают требования $\mathcal{I}_{\textbf{mem1}}$.

Проверим, что выполняется $\mathcal{I}_{\textbf{mem2}}(\mathbf{a}', \mathbf{p}')$. Для этого нужно показать, что верно следующее утверждение.

$$
\begin{array}{l}
\forall \langle y : val'@\tau', view''\rangle \in \mathbf{p}'.M, \tau' \neq \mathbf{0} \Rightarrow\\
\quad \exists tid', path'', view''' \geqslant view'',\\
\qquad \mathsf{W}\ (\mathsf{com}\ \_\ y\ val') = \mathbf{a}'.tapef(tid', path''), \mathbf{a}'.H(tid', path'') = (\tau', \_, view''').
\end{array}
$$

Зафиксируем сообщение $\langle y : val'@\tau', view''\rangle$ такое, что $\tau' \neq \mathbf{0}$.

$$
\begin{array}{l}
\langle y : val'@\tau', view''\rangle \in \mathbf{p}.M \cup \{\langle x : val@\tau, view'\rangle\} \Rightarrow\\
\quad \exists tid', path'', view''' \geqslant view'',\\
\qquad \mathsf{W}\ (\mathsf{com}\ \_\ y\ val') = \mathbf{a}'.tapef(tid', path''), \mathbf{a}'.H(tid', path'') = (\tau', \_, view''').
\end{array}
$$

Если $\langle y : val'@\tau', view''\rangle \in \mathbf{p}.M$, то утверждение является следствием утверждения $\mathcal{I}_{\textbf{mem2}}(\mathbf{a}, \mathbf{p})$. Иначе, $\langle y : val'@\tau', view''\rangle$ совпадает с $\langle x : val@\tau, view'\rangle$. Тогда мы

можем упростить утверждение:

$$\exists tid', path'', view''' \geqslant view', \mathsf{W}\ (\mathsf{com}\ \_\ x\ val) = \mathbf{a}'.tapef(tid, path''),$$
$$\mathbf{a}'.H(tid', path'') = (\tau, \_, view''').$$

Выберем $tid' = tid, path'' = path, view''' = view$. Тогда утверждение верно по определению $\mathbf{a}'$. □

## B.2 Сертификация

При доказательстве того, что обещающая машина может симулировать исполнение ARM+τ машины, мы используем леммы 5 и 7, в которых конструируется шаг обещающей машины. Согласно определению, после каждого шага обещающая машина должна показывать, что находится в *сертифицируемом состоянии* (предикат certifiable), т.е. что для каждого потока существует последовательность локальных шагов, при исполнении которых поток выполняет все данные обещания (предикат $\mathsf{certifiable}_{tid}$):

$$\begin{aligned}
\mathsf{certifiable}(\mathbf{p}) &\triangleq \forall tid.\ \mathsf{certifiable}_{tid}(\mathsf{thread\text{-}state}(tid, \mathbf{p}));\\
\mathsf{certifiable}_{tid}(\mathbf{t}) &\triangleq \exists \mathbf{t}'.\ \mathbf{t} \xrightarrow[\mathsf{Promise}\ tid]{}{}^{*}\ \mathbf{t}' \wedge \mathbf{t}'.promises = \varnothing;\\
\mathsf{thread\text{-}state}(tid, \mathbf{p}) &\triangleq \langle \mathbf{p}.M, \mathbf{p}.\mathcal{TS}(tid) \rangle.
\end{aligned}$$

Теорема 7, которая является ключевой в данном приложении и используется в доказательстве лемм 5 и 7, утверждает, что если состояния обещающей и ARM+τ машин связаны отношением $\mathcal{I}_{\text{base}}$, и при этом состояние ARM+τ машины достижимо из начального, то состояние обещающей машины сертифицируемо.

**Теорема 7.** $\forall (\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{base}}.\ \mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*}\ \mathbf{a} \Rightarrow \mathsf{certifiable}(\mathbf{p})$.

### B.2.1 Структура доказательства теоремы о сертификации

Какие невыполненные обещания есть у потока *tid* обещающей машины в состоянии **p**, если оно связано с некоторым состоянием **a** машины ARM+τ отношением $\mathcal{I}_{\text{base}}$? Согласно отношению $\mathcal{I}_{\mathbf{mem2}} \subseteq \mathcal{I}_{\text{base}}$ для каждого сообщения (в том числе обещанного, но не выполненного) в памяти обещающей машины в плёнке машины ARM+τ существует завершённый экземпляр инструкции записи, ему соответствующий. Из отношения $\mathcal{I}_{\mathbf{mem1}} \subseteq \mathcal{I}_{\text{base}}$ следует, что обещанное, но не выполненное сообщение соответствует экземпляру, чей путь не меньше,

чем текущий указатель потока обещающей машины. Таким образом, для сертификации поток обещающей машины может совершить серию переходов, которые покроют плёнку машины ARM+τ вплоть до последнего завершённого экземпляра чтения. При этом обещающая машина будет выполнять переходы, которые будут определены предшествующими экземплярами инструкций в состоянии машины ARM+τ. Именно наличие такой серии переходов мы будем доказывать.

Для реализации описанной выше идеи мы используем отношение $\mathcal{I}_{\text{cert}}(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t})$, которое имеет шесть параметров:

- *tid* — это идентификатор потока, в контексте сертификации которого используется данный элемент отношения;
- $n$ — это количество экземпляров на пути от указателя потока до последнего завершённого экземпляра записи в соответствующей плёнке;
- $k$ — это номер инструкции, следующей за последним завершённым экземпляром записи;
- $\delta$ — это частичная функция, имеющая тип $Path \rightharpoonup (\texttt{Time} \times \mathcal{V})$, которая экземпляру не завершённой инструкции записи в плёнке потока *tid* ставит в соответствие пару из метки времени и фронта сообщения, которое было добавлено в память обещающей машиной;
- **a** — это состояние машины ARM+τ, которое используется для сертификации потока обещающей машины;
- **t** — это состояние потока *tid* обещающей машины.

Перед тем, как предъявить формальное определение данного отношения, мы приведём утверждения лемм, которые его используют.

Лемма 12 утверждает, что из $(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{cert}}$ следует, что поток *tid* обещающей машины сертифицируем.

**Лемма 12.** $\forall n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}.\ \mathcal{I}_{\text{cert}}(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}) \Rightarrow \mathsf{certifiable}_{\mathit{tid}}(\mathbf{t})$.

Доказательство данной леммы проводится индукцией по $n$. Базу индукции мы выделили в лемму 13, а индукционный переход — в лемму 14.

**Лемма 13.** $\forall k, \delta, \mathit{tid}, \mathbf{a}, \mathbf{t}, \mathcal{I}_{\text{cert}}(0, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}), \mathbf{t}.\mathit{promises} = \varnothing$.

**Лемма 14.** $\forall n \neq 0, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}, \mathcal{I}_{\text{cert}}(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t})$.
$\exists \delta', \mathbf{t}'.\mathit{Prog}(\mathit{tid}) \vdash \mathbf{t} \xrightarrow[\text{Promise } \mathit{tid}]{}{}^{*} \mathbf{t}' \wedge \mathcal{I}_{\text{cert}}(n-1, \delta', k, \mathit{tid}, \mathbf{a}, \mathbf{t}')$.

### B.2.2 Описание вспомогательного отношения

Отношение $\mathcal{I}_{\text{cert}}$ определяется следующим образом.

$$
\begin{aligned}
&\mathcal{I}_{\text{cert}} \subset \mathbb{N} \times (Path \rightharpoonup (\texttt{Time} \times \mathcal{V})) \times \mathbb{N} \times Tid \times \mathsf{State}_{ARM+\tau} \times \mathsf{TState}_{\mathsf{Promise}} \\
&\mathcal{I}_{\text{cert}}(n, \delta, k, tid, \mathbf{a}, \mathbf{t}) \triangleq \\
&\quad \underline{\text{let}}\ tape \triangleq \mathbf{a}.tapef(tid)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ path_{\mathsf{last\text{-}wcom}} \triangleq \mathsf{last\text{-}write\text{-}com}(tape)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ path_{\mathsf{next\text{-}last}} \triangleq path_{\mathsf{last\text{-}wcom}} : k\ \underline{\text{in}} \\
&\quad \mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{a} \wedge \\
&\quad \mathbf{t}.path \leqslant path_{\mathsf{next\text{-}last}} \wedge (n = \mathsf{length}(path_{\mathsf{next\text{-}last}}) - \mathsf{length}(\mathbf{t}.path)) \wedge \\
&\quad (\forall path' \geqslant \mathbf{t}.path, \delta(path') = \bot) \wedge \\
&\qquad path'' \geqslant path' \geqslant \mathbf{t}.path \wedge tape(path') = \mathsf{R}\ st_{\text{read}} \wedge st_{\text{read}} \neq \mathsf{sat\ com}\ \_ \Rightarrow \\
&\qquad\quad tape(path'') \neq \mathsf{F}\ \_\ \_) \wedge \\
&\quad (\forall path'.\ \mathbf{t}.path \leqslant path' < path_{\mathsf{next\text{-}last}} \Rightarrow \\
&\qquad (Prog(tid)[path'.\mathsf{last}] \in \{\text{``}\mathsf{if}\ \_\ \mathsf{goto}\ \_\text{''}, \text{``}\mathsf{fence}(\_)\text{''}\} \Rightarrow tape(path') \text{ завершён}) \wedge \\
&\qquad tape(path') \text{ имеет полностью определённый адрес} \wedge \\
&\qquad \neg Prog(tid)[path'.\mathsf{last}] = \text{``}\mathsf{fence}(\texttt{sy})\text{''}) \wedge \\
&\quad (\forall path'', path', st_{\text{read}}. \\
&\quad (tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-com-cert}} \cap \mathcal{I}_{\text{mem-2-cert}} \cap \mathcal{I}_{\text{state-cert}} \cap \mathcal{I}_{\text{write-rel-cert}} \cap \mathcal{I}_{\text{view-write-cert}} \wedge \\
&\quad (\delta, tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-tid-cert}} \cap \mathcal{I}_{\delta\text{-con-1}} \cap \mathcal{I}_{\delta\text{-con-2}} \cap \mathcal{I}_{\text{view-read-cert}} \wedge \\
&\quad (\delta, tid, \mathbf{a}) \in \mathcal{I}_{\delta\text{-con-3}} \cap \mathcal{I}_{\delta\text{-con-4}}.
\end{aligned}
$$

В определении используются отношения, определённые ниже.

$$
\begin{aligned}
&\mathcal{I}^{tid}_{\mathsf{w-cert}}(tid, \mathbf{a}, \mathbf{p}) \triangleq \\
&\quad \underline{\text{let}}\ tape, path \triangleq \mathbf{a}.tapef(tid), \mathbf{p}.\mathcal{TS}(tid).path\ \underline{\text{in}} \\
&\quad \exists path' \geqslant path.tape(path') \text{ является завершённой записью.}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{I}_{\text{write-rel-cert}}(tid, \mathbf{a}, \mathbf{t}) \triangleq \forall path \geqslant \mathbf{t}.path. \\
&\quad \underline{\text{let}}\ tape \triangleq \mathbf{a}.tapef(tid)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ path^{\texttt{ld}} \triangleq \mathsf{last}\texttt{ld}(tape, path)\ \underline{\text{in}} \\
&\quad \mathbf{t}.view_{\text{rel}} \leqslant \bigsqcup \mathsf{com\text{-}reads\text{-}view}(path^{\texttt{ld}}, tape, \mathbf{a}.H) \sqcup \\
&\qquad \bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid, path, tape, \mathbf{a}.H).
\end{aligned}
$$

$$\mathcal{I}_{\text{view-write-cert}}(tid, \mathbf{a}, \mathbf{t}) \triangleq \forall path' \geqslant \mathbf{t}.path, \ell.\ \mathbf{a}.tapef(tid, path') = \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ \_) \Rightarrow$$
$$\mathbf{t}.view_{\text{cur}}(\ell) < \mathbf{a}.H_\tau(tid, path') \wedge$$
$$((\exists path''.\ \mathbf{t}.path \leqslant path'' < path' \wedge \mathbf{a}.tapef(tid, path'') = \mathsf{F}\ \mathsf{com}\ \texttt{ld}) \Rightarrow$$
$$\mathbf{t}.view_{\text{acq}}(\ell) < \mathbf{a}.H_\tau(tid, path')).$$

$$\mathsf{comb\text{-}time}(\delta, tid_{\mathbf{t}}, \mathbf{a}) \triangleq \lambda tid, path.$$
$$\underline{\text{if}}\ tid = tid_{\mathbf{t}} \wedge \mathbf{a}.H(tid, path) = \bot\ \underline{\text{then}}\ \delta(path)$$
$$\underline{\text{elif}}\ \exists \tau, view, \langle \tau, \_, view\rangle = \mathbf{a}.H_\tau(tid, path)\ \underline{\text{then}}\ \langle \tau, view\rangle$$
$$\underline{\text{else}}\ \bot.$$

$$\mathcal{I}_{\text{view-read-cert}}(\delta, tid, \mathbf{a}, \mathbf{t}) \triangleq \forall path' \geqslant \mathbf{t}.path, w, \tau.$$
$$\mathbf{a}.tapef(tid, path') = \mathsf{R}\ (\mathsf{sat}\ \mathsf{com}\ w) \wedge \langle \tau, \_\rangle = \mathsf{comb\text{-}time}(\delta, tid, \mathbf{a}, w.tid, w.path) \Rightarrow$$
$$\mathbf{t}.view_{\text{cur}}(w.\ell) \leqslant \tau \wedge$$
$$((\exists path''.\ \mathbf{t}.path \leqslant path'' < path' \wedge \mathbf{a}.tapef(tid, path'') = \mathsf{F}\ \mathsf{com}\ \texttt{ld}) \Rightarrow$$
$$\mathbf{t}.view_{\text{acq}}(w.\ell) \leqslant \tau).$$

$$\mathcal{I}_{\text{state-cert}}(tid, \mathbf{a}, \mathbf{t}) \triangleq$$
$$\underline{\text{let}}\ \mathsf{regf}_{\mathsf{com}} \triangleq \mathsf{regf}_{\mathsf{com}}(Prog(tid), \mathbf{a}.tapef(tid), \mathbf{t}.path)\ \underline{\text{in}}$$
$$\forall reg.\ \mathsf{regf}_{\mathsf{com}}(reg) = \bot \vee \mathbf{t}.st(reg) = \mathsf{regf}_{\mathsf{com}}(reg).$$

$$\mathcal{I}_{\text{mem-1-tid-cert}}(\delta, tid_{\mathbf{t}}, \mathbf{a}, \mathbf{t}) \triangleq \forall path, \tau, view, \ell, expr_0, expr_1.$$
$$\langle \tau, view\rangle = \delta(path) \wedge \text{“}[expr_0]\ :=\ expr_1\text{”} = Prog(tid_{\mathbf{t}}, path.\mathsf{last}) \wedge \ell = [\![expr_0]\!]^{path}_{\mathsf{com}} \Rightarrow$$
$$\exists val.[\![expr_1]\!]^{path}_{\mathsf{com}} \in \{\bot, val\} \wedge \langle \ell : val@\tau, view\rangle \in \mathbf{t}.M \setminus \mathbf{t}.promises.$$

$$\mathcal{I}_{\text{mem-1-com-cert}}(tid_{\mathbf{t}}, \mathbf{a}, \mathbf{t}) \triangleq \forall tid, \ell, val, \tau, view', path.$$
$$\mathsf{W}\ (\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \langle \tau, \_, view'\rangle = \mathbf{a}.H(tid, path) \Rightarrow$$
$$\exists view \leqslant view'.\ \langle \ell : val@\tau, view\rangle \in \mathbf{t}.M \wedge$$
$$(tid \neq tid_{\mathbf{t}} \vee path < \mathbf{t}.path \Rightarrow \langle \ell : val@\tau, view\rangle \notin \mathbf{t}.promises) \wedge$$
$$(tid = tid_{\mathbf{t}} \wedge path \geqslant \mathbf{t}.path \Rightarrow \langle \ell : val@\tau, view\rangle \in \mathbf{t}.promises).$$

$$\mathcal{I}_{\text{mem-2-cert}}(tid, \mathbf{a}, \mathbf{t}) \triangleq \forall \langle \ell : val@\tau, view\rangle \in \mathbf{t}.promises.\ \tau \neq \mathbf{0} \Rightarrow$$
$$\exists view' \geqslant view, path \geqslant \mathbf{t}.path.$$
$$\mathsf{W}\ (\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \langle \tau, \_, view'\rangle = \mathbf{a}.H(tid, path).$$

$\mathcal{I}_{\delta\text{-con-1}}(\delta, tid_{\mathbf{t}}, \mathbf{a}, \mathbf{t}) \triangleq \forall path < \mathbf{t}.path.\ \delta(path) \neq \bot \Leftrightarrow$
$(\exists expr_0, expr_1.\ \text{``}[expr_0]\ :=\ expr_1\text{''} = Prog(tid_{\mathbf{t}}, path.\mathsf{last}) \wedge$
$\mathbf{a}.tapef(tid_{\mathbf{t}}, path)$ isn't committed).

$\mathcal{I}_{\delta\text{-con-2}}(\delta, tid_{\mathbf{t}}, \mathbf{a}, \mathbf{t}) \triangleq \forall path, \langle \tau, view \rangle = \delta(path),$
$\text{``}[expr_0]\ :=\ expr_1\text{''} = Prog(tid_{\mathbf{t}}, path.\mathsf{last}), \ell = [\![expr_0]\!]^{path}_{\mathsf{com}}.$
$view = [\ell @ \tau] \sqcup \mathbf{t}.view_{\text{rel}} \wedge \mathbf{t}.view_{\text{cur}}(\ell) \geqslant \tau.$

$\mathcal{I}_{\delta\text{-con-3}}(\delta, tid_{\mathbf{t}}, \mathbf{a}) \triangleq \forall path, path' \neq path,$
$\langle \tau, \_ \rangle = \mathsf{comb\text{-}time}(\delta, tid_{\mathbf{t}}, \mathbf{a}, tid_{\mathbf{t}}, path), \langle \tau', \_ \rangle = \mathsf{comb\text{-}time}(\delta, tid_{\mathbf{t}}, \mathbf{a}, tid_{\mathbf{t}}, path'),$
$\text{``}[expr_0]\ :=\ expr_1\text{''} = Prog(tid_{\mathbf{t}}, path.\mathsf{last}),$
$\text{``}[expr'_0]\ :=\ expr'_1\text{''} = Prog(tid_{\mathbf{t}}, path'.\mathsf{last}).$
$[\![expr_0]\!]^{path}_{\mathsf{com}} = [\![expr'_0]\!]^{path'}_{\mathsf{com}} \Rightarrow \tau \neq \tau'.$

$\mathcal{I}_{\delta\text{-con-4}}(\delta, tid_{\mathbf{t}}, \mathbf{a}) \triangleq \forall path_\delta < path_{read} < path_{\mathtt{ld}} < path_{\delta - read},$
$\langle \tau, \_ \rangle = \delta(path_\delta), w, \ell, view.$
$tape(path_{read}) = \mathsf{R}\ (\mathsf{sat\ com}\ w) \wedge tape(path_{\mathtt{ld}}) = \mathsf{F\ com}\ \mathtt{ld} \wedge$
$tape(path_{\delta - read}) = \mathsf{R}\ (\mathsf{sat\ com}\ \langle tid_{\mathbf{t}}, path_\delta, \mathsf{wr}\ \ell : \_ \rangle) \wedge$
$view = \mathbf{a}.H_{\mathsf{view}}(w.tid, w.path) \neq \bot \Rightarrow$
$view(\ell) \leqslant \tau.$

### B.2.3 Доказательство лемм и теоремы о сертификации

**Лемма 13.** $\forall k, \delta, tid, \mathbf{a}, \mathbf{t}, \mathcal{I}_{\text{cert}}(0, \delta, k, tid, \mathbf{a}, \mathbf{t}), \mathbf{t}.promises = \varnothing.$

*Доказательство.* Зафиксируем $k, \delta, tid, \mathbf{a}, \mathbf{t}$. Поскольку $\mathcal{I}_{\text{cert}}(0, \delta, k, tid, \mathbf{a}, \mathbf{t})$ выполняется и верно, что $\mathsf{length}(\mathbf{t}.path) = \mathsf{length}(\mathsf{last\text{-}write\text{-}com}(\mathbf{a}.tapef(tid))) + 1$, то $\mathbf{t}.path = \mathsf{last\text{-}write\text{-}com}(\mathbf{a}.tapef(tid)) : k$.

Предположим, что существует ещё не выполненное обещание, т.е. $\exists \langle \ell : val@\tau, view \rangle \in \mathbf{t}.promises$. Тогда, из $\mathcal{I}_{\text{mem-2-cert}}(tid, \mathbf{a}, \mathbf{t})$ следует, что существует фронт $view' \geqslant view$ и путь $path \geqslant \mathbf{t}.path$ такие, что $\mathsf{W}\ (\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path)$. А значит $path \geqslant \mathbf{t}.path > \mathsf{last\text{-}write\text{-}com}(\mathbf{a}.tapef(tid))$. Т.о. мы получили противоречие с определением $\mathsf{last\text{-}write\text{-}com}$. □

**Лемма 14.** $\forall n \neq 0, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t}, \mathcal{I}_{\text{cert}}(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t})$.
$\exists \delta', \mathbf{t}'. \mathit{Prog}(\mathit{tid}) \vdash \mathbf{t} \xrightarrow[\text{Promise } \mathit{tid}]{}{}^{*} \mathbf{t}' \wedge \mathcal{I}_{\text{cert}}(n-1, \delta', k, \mathit{tid}, \mathbf{a}, \mathbf{t}')$.

*Доказательство.* Зафиксируем $n, \mathit{tid}, \mathbf{a}, \mathbf{t}$. Введём следующие обозначения:

$$\begin{array}{lcl} \mathit{tape} & \triangleq & \mathbf{a}.\mathit{tapef}(\mathit{tid}); \\ \langle M, \langle \mathit{path}, \mathit{st}, V, \mathit{promises} \rangle \rangle & \triangleq & \mathbf{t}; \\ \mathit{cmds} & \triangleq & \mathit{Prog}(\mathit{tid}). \end{array}$$

Далее в доказательстве нам нужно рассмотреть варианты $\mathit{tape}(\mathit{path})$. Варианты, при которых $\mathit{tape}(\mathit{path}) \in \{\mathsf{Nop}, \mathsf{Assign}, \mathsf{If}\ \_\ \_, \mathit{tape}(\mathit{path}) = \mathsf{F}\ \_\ \_\}$, проверяются тривиальным образом (в этих случаях $\delta' = \delta$). Рассмотрим оставшиеся варианты.

– $\mathit{tape}(\mathit{path}) = \mathsf{R}\ (\mathsf{sat}\ \mathsf{com}\ w)$. Поскольку $\mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{a}$ и $\mathit{tape}(\mathit{path}) = \mathsf{R}\ (\mathsf{sat}\ \mathsf{com}\ w)$, то $\mathit{cmds}(\mathit{path}.\mathsf{last}) =$ ‘‘$\mathit{reg}\ :=\ [\mathit{expr}]$”. Введём следующие обозначения:

$$\begin{array}{lcl} \mathit{path}' & \triangleq & \mathsf{next\text{-}path}(\mathit{path}, 1); \\ \ell & \triangleq & [\![\mathit{expr}]\!]^{\mathit{path}}_{\mathsf{com}} = w.\ell = [\![\mathit{expr}]\!]^{\mathit{st}}; \\ \mathit{val} & \triangleq & w.\mathit{val}; \\ \mathit{st}' & \triangleq & \mathit{st}[\mathit{reg} \mapsto \mathit{val}]. \end{array}$$

Поскольку верно $\mathcal{I}_{\text{view-read-cert}}(\delta, \mathit{tid}, \mathbf{a}, \mathbf{t})$, то существуют $\tau$ и $\mathit{view}$ такие, что $(\tau, \mathit{view}) = \mathsf{comb\text{-}time}(\delta, \mathbf{a}, w.\mathit{tid}, w.\mathit{path})$ и $\mathit{view}_{\text{cur}}(\ell) \leqslant \tau$.
То, что $\langle \ell : \mathit{val}@\tau, \mathit{view} \rangle \in \mathbf{t}.M \setminus \mathbf{t}.\mathit{promises}$, следует из $(\delta, \mathit{tid}, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-com-cert}} \cap \mathcal{I}_{\text{mem-1-com-cert}}$. Введём обозначение:

$$V' = \langle \mathit{view}'_{\text{cur}}, \mathit{view}'_{\text{acq}}, \mathit{view}'_{\text{rel}} \rangle \triangleq \langle \mathit{view}_{\text{cur}} \sqcup [\ell@\tau], \mathit{view}_{\text{acq}} \sqcup \mathit{view}, \mathit{view}_{\text{rel}} \rangle.$$

Тогда $\mathbf{t} \xrightarrow[\text{Promise } \mathit{tid}]{\textbf{Read from memory } \langle \ell:\mathit{val}@\tau,\mathit{view} \rangle} \mathbf{t}' \triangleq \langle M, \langle \mathit{path}', V', \mathit{promises} \rangle \rangle$.
То, что выполняется $\mathcal{I}_{\text{cert}}(n-1, \delta, k, \mathbf{a}, \mathbf{t}')$, проверяется тривиальным образом.

– $\mathit{tape}(\mathit{path}) = \mathsf{R}\ \mathit{st}_{\text{read}}$, где состояние $\mathit{st}_{\text{read}}$ указывает, что экземпляр не завершён. Аналогично предыдущему пункту, $\mathit{cmds}(\mathit{path}.\mathsf{last}) =$ ‘‘$\mathit{reg}\ :=$

$[expr]$". Введём следующие обозначения:

$$\begin{array}{lcl}
path' & \triangleq & \mathsf{next\text{-}path}(path, 1); \\
\mathsf{regf}_{\mathsf{com}} & \triangleq & \mathsf{regf}_{\mathsf{com}}(cmds, tape, path); \\
\mathsf{regf}'_{\mathsf{com}} & \triangleq & \mathsf{regf}_{\mathsf{com}}(cmds, tape, path') \\
& = & \mathsf{regf}_{\mathsf{com}}(cmds, tape, path)[reg \mapsto \bot] \text{ (by definition)}; \\
\ell & \triangleq & [\![expr]\!]^{path}_{\mathsf{com}} \\
& = & [\![expr]\!]^{st} \text{ (by } \mathcal{I}_{\text{state-cert}}(tid, \mathbf{a}, \mathbf{t})); \\
\tau & \triangleq & view_{\text{cur}}(\ell).
\end{array}$$

Из свойств обещающей машины следует, что существуют *val* и *view* такие, что $\langle \ell : val@\tau, view \rangle \in M$. Введём следующие обозначения:

$$\begin{array}{lcl}
V' = \langle view'_{\text{cur}}, view'_{\text{acq}}, view'_{\text{rel}} \rangle & \triangleq & \langle view_{\text{cur}} \sqcup [\ell@\tau], view_{\text{acq}} \sqcup view, view_{\text{rel}} \rangle \\
& = & \langle view_{\text{cur}}, view_{\text{acq}} \sqcup view, view_{\text{rel}} \rangle; \\
st' & \triangleq & st[reg \mapsto val].
\end{array}$$

По определению, выполняется $\mathbf{t} \xrightarrow[\mathsf{Promise}\ tid]{\textbf{Read from memory } \langle \ell : val@\tau, view \rangle} \mathbf{t}' \triangleq \langle M, \langle path', st', V', promises \rangle \rangle$. Проверка $\mathcal{I}_{\text{cert}}(n-1, \delta, k, \mathbf{a}, \mathbf{t}')$ также тривиальна.

– $tape(path) = \mathsf{W}\ (\mathsf{com}\ im\ \ell\ val)$. Аналогично предыдущим пунктам, $cmds(path.\mathsf{last}) =$ "$[expr_0]\ :=\ expr_1$". Введём следующие обозначения:

$$\begin{array}{lcl}
path' & \triangleq & \mathsf{next\text{-}path}(path, 1); \\
\mathsf{regf}_{\mathsf{com}} & \triangleq & \mathsf{regf}_{\mathsf{com}}(cmds, tape, path); \\
\langle \tau, _, view' \rangle & \triangleq & \mathbf{a}.H(tid, path).
\end{array}$$

Из $\mathcal{I}_{\text{state-cert}}(tid, \mathbf{a}, \mathbf{t})$ следует, что $\ell = [\![expr_0]\!]^{path}_{\mathsf{com}} = [\![expr_0]\!]^{st}$ и $val = [\![expr_1]\!]^{path}_{\mathsf{com}} = [\![expr_1]\!]^{st}$. Из $\mathcal{I}_{\text{mem-1-com-cert}}(tid, \mathbf{a}, \mathbf{t})$ следует, что существует $view \leqslant view'$ такой, что $\langle \ell : val@\tau, view \rangle \in \mathbf{t}.promises \cap \mathbf{t}.M$. То, что $view_{\text{cur}}(\ell) < \tau$, следует из $\mathcal{I}_{\text{view-write-cert}}(tid, \mathbf{a}, \mathbf{t})$.

$$\begin{array}{lcl}
V' & \triangleq & \langle view_{\text{cur}} \sqcup [\ell@\tau], view_{\text{acq}} \sqcup [\ell@\tau], view_{\text{rel}} \rangle; \\
promises' & \triangleq & promises \setminus \{\langle \ell : val@\tau, view \rangle\}.
\end{array}$$

Так, выполняется $\mathbf{t} \xrightarrow[\mathsf{Promise}\ tid]{\textbf{Fulfill promise } \langle \ell : val@\tau, view \rangle} \mathbf{t}' \triangleq \langle M, \langle path', st, V', promises' \rangle \rangle$. Проверка $\mathcal{I}_{\text{cert}}(n-1, \delta, k, \mathbf{a}, \mathbf{t}')$ тривиальна.

– $tape(path) = \mathsf{W}\ st_{\text{write}}$, где состояние $st_{\text{write}}$ указывает, что экземпляр не завершён. В этом случае поток обещающей машины с идентификатором *tid* совершит два перехода: сначала пообещает некоторое сообщение, а потом выполнит это обещание. Аналогично предыдущим пунктам, $cmds(path.\mathsf{last}) = \text{``}[expr_0]\ :=\ expr_1\text{''}$. Введём следующие обозначения:

$$\begin{array}{lcl} path' & \triangleq & \mathsf{next\text{-}path}(path, 1); \\ \mathsf{regf}_{\mathsf{com}} & \triangleq & \mathsf{regf}_{\mathsf{com}}(cmds, tape, path); \\ \ell & \triangleq & [\![expr_0]\!]^{path}_{\mathsf{com}} = [\![expr_0]\!]^{st}; \\ val & \triangleq & [\![expr_1]\!]^{st}. \end{array}$$

Поскольку соответствующий экземпляр записи не завершён, то в памяти ARM+τ машины нет связанного с ним запроса, а значит, и нужной метки времени для добавляемого обещающей машиной сообщения. Поэтому нам нужно выбрать эту метку времени, которую мы будем обозначать τ. Метка τ должна удовлетворять следующим ограничениям.

1. $view_{\text{cur}}(\ell) < \tau$.
2. $\tau \notin \{\tau' | \langle \ell, \_, \mathsf{wr}\ \tau' : \_ \rangle \in M\}$.
3. $\forall path_{write} \geqslant path.\ tape(path_{write}) = \mathsf{W}\ (\mathsf{com}\ \_\ \ell\ \_) \Rightarrow$
   $\tau < \mathbf{a}.H_\tau(tid, path_{write})$.
4. $\forall path_{read} \geqslant path.\ tape(path_{read}) = \mathsf{R}\ (\mathsf{sat\ com}\ w) \wedge$
   $\mathbf{a}.H_\tau(w.tid, w.path) \neq \bot \Rightarrow$
   $\tau \leqslant \mathbf{a}.H_\tau(w.tid, w.path)$.
5. $\forall path_{read} < path_{\texttt{ld}} < path_{\delta-read}, w, view.path < path_{read} \wedge$
   $tape(path_{read}) = \mathsf{R}\ (\mathsf{sat\ com}\ w) \wedge tape(path_{\texttt{ld}}) = \mathsf{F\ com}\ \texttt{ld} \wedge$
   $tape(path_{\delta-read}) = \mathsf{R}\ (\mathsf{sat\ com}\ \langle tid, path, \mathsf{wr}\ \ell : \_ \rangle) \wedge$
   $view = \mathbf{a}.H_{\mathsf{view}}(w.tid, w.path) \neq \bot \Rightarrow$
   $view(\ell) \leqslant \tau$.

Поскольку метки времени являются элементами плотного множества $\mathbb{Q}$, то в силу $\mathcal{I}_{\text{view-write-cert}}(tid, \mathbf{a}, \mathbf{t})$, $\mathcal{I}_{\text{view-read-cert}}(\delta, tid, \mathbf{a}, \mathbf{t})$ и теоремы 4 нужная метка τ существует.

Введём следующие обозначения:

$$
\begin{array}{lcl}
\mathit{view} & \triangleq & \mathit{view}_{\text{rel}} \sqcup [\ell@\tau]; \\
msg & \triangleq & \langle \ell : \mathit{val}@\tau, \mathit{view} \rangle; \\
M' & \triangleq & M \cup \{msg\}; \\
V' = \langle \mathit{view}'_{\text{cur}}, \mathit{view}'_{\text{acq}}, \mathit{view}'_{\text{rel}} \rangle & \triangleq & \langle \mathit{view}_{\text{cur}} \sqcup [\ell@\tau], \mathit{view}_{\text{acq}} \sqcup [\ell@\tau], \mathit{view}_{\text{rel}} \rangle; \\
\delta' & \triangleq & \delta[\mathit{path} \mapsto \langle \tau, \mathit{view} \rangle].
\end{array}
$$

Тогда, верно следующее:

$$
\mathbf{t} \xrightarrow[\substack{\text{Promise write } \langle \ell:\mathit{val}@\tau,\mathit{view}\rangle \\ \text{Promise } \mathit{tid}}]{} \langle M', \langle \mathit{path}, \mathit{st}, V, \mathit{promises} \cup \{msg\} \rangle \rangle
$$
$$
\xrightarrow[\substack{\text{Fulfill promise } \langle \ell:\mathit{val}@\tau,\mathit{view}\rangle \\ \text{Promise } \mathit{tid}}]{} \mathbf{t}' \triangleq \langle M', \langle \mathit{path}', \mathit{st}, V', \mathit{promises} \rangle \rangle.
$$

Проверка $\mathcal{I}_{\text{cert}}(n-1, \delta', k, \mathit{tid}, \mathbf{a}, \mathbf{t}')$ тривиальна. □

**Теорема 7.** $\forall (\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{base}}.\mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{a} \Rightarrow \mathsf{certifiable}(\mathbf{p})$.

*Доказательство.* Зафиксируем $\mathit{tid}, \mathit{tape} \triangleq \mathbf{a}.\mathit{tapef}(\mathit{tid})$. Введём следующие обозначения:

$$
\begin{array}{lcl}
k & \triangleq & \mathsf{last\text{-}write\text{-}com}(\mathit{tape}).\mathsf{last} + 1; \\
n & \triangleq & \mathsf{length}(\mathsf{last\text{-}write\text{-}com}(\mathit{tape}) : k) - \mathsf{length}(\mathbf{t}.\mathit{path}); \\
\mathit{path}_{\mathsf{last\text{-}wcom}} & \triangleq & \mathsf{last\text{-}write\text{-}com}(\mathit{tape}); \\
\mathit{path}_{\mathsf{next\text{-}last}} & \triangleq & \mathit{path}_{\mathsf{last\text{-}wcom}} : k; \\
\delta & \triangleq & \lambda \mathit{path}.\ \bot.
\end{array}
$$

Если мы сможем показать, что $\mathcal{I}_{\text{cert}}(n, \delta, k, \mathit{tid}, \mathbf{a}, \mathbf{t})$ выполняется, то утверждение теоремы верно по лемме 12. Таким образом нам нужно проверить следующее

утверждение:

$$
\begin{aligned}
&\mathbf{a}^{\text{init}} \xrightarrow[\text{ARM}]{}{}^{*} \mathbf{a} \wedge \\
&\mathbf{t}.path \leqslant path_{\text{next-last}} \wedge (n = \mathsf{length}(path_{\text{next-last}}) - \mathsf{length}(\mathbf{t}.path)) \wedge \\
&(\forall path' \geqslant \mathbf{t}.path, \delta(path') = \bot) \wedge \\
&(\forall path'', path', st_{\text{read}}. \\
&\quad path'' \geqslant path' \geqslant \mathbf{t}.path \wedge tape(path') = \mathsf{R}\ st_{\text{read}} \wedge st_{\text{read}} \neq \mathsf{sat\ com}\ \_ \Rightarrow \\
&\qquad tape(path'') \neq \mathsf{F}\ \_\ \_) \wedge \\
&(\forall path'.\ \mathbf{t}.path \leqslant path' < path_{\text{next-last}} \Rightarrow \\
&\quad (Prog(tid)[path'.\mathsf{last}] \in \{\text{``}\mathsf{if}\ \_\ \mathsf{goto}\ \_\text{''}, \text{``}\mathsf{fence}(\_)\text{''}\} \Rightarrow tape(path') \text{ завершён}) \wedge \\
&\quad tape(path') \text{ имеет полностью определённый адрес} \wedge \\
&\quad \neg Prog(tid)[path'.\mathsf{last}] = \text{``}\mathsf{fence}(\mathtt{sy})\text{''}) \wedge \\
&(tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-com-cert}} \cap \mathcal{I}_{\text{mem-2-cert}} \cap \mathcal{I}_{\text{state-cert}} \cap \mathcal{I}_{\text{write-rel-cert}} \cap \mathcal{I}_{\text{view-write-cert}} \wedge \\
&(\delta, tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-tid-cert}} \cap \mathcal{I}_{\delta\text{-con-1}} \cap \mathcal{I}_{\delta\text{-con-2}} \cap \mathcal{I}_{\text{view-read-cert}} \wedge \\
&(\delta, tid, \mathbf{a}) \in \mathcal{I}_{\delta\text{-con-3}} \cap \mathcal{I}_{\delta\text{-con-4}}.
\end{aligned}
$$

Первые три конъюнкта очевидно выполняются. Четвертый конъюнкт выполняется в силу ограничений перехода **Завершение барьера** машины ARMv8 POP. Пятый конъюнкт выполняется по ограничениям перехода **Завершение записи** машины ARMv8 POP и истинности утверждения $\mathcal{I}_{\text{com-SY}}(\mathbf{a}, \mathbf{p})$.

Утверждения $(\delta, tid, \mathbf{a}) \in \mathcal{I}_{\delta\text{-con-3}} \cap \mathcal{I}_{\delta\text{-con-4}}$ и $(\delta, tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\delta\text{-con-1}} \cap \mathcal{I}_{\delta\text{-con-2}} \cap \mathcal{I}_{\text{mem-1-tid-cert}}$ верны, поскольку $\delta = \lambda path.\ \bot$ является нигде не определённой функцией. Утверждение $(\delta, tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{view-read-cert}}$ верно, поскольку $\delta = \lambda path.\ \bot$ и выполняется теорема 4.

Утверждение $(tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{mem-1-com-cert}} \cap \mathcal{I}_{\text{mem-2-cert}} \cap \mathcal{I}_{\text{state-cert}}$ непосредственно следует из $(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\mathbf{mem1}} \cap \mathcal{I}_{\mathbf{mem2}} \cap \mathcal{I}_{\mathbf{state}}$. Утверждение $(tid, \mathbf{a}, \mathbf{t}) \in \mathcal{I}_{\text{write-rel-cert}} \cap \mathcal{I}_{\text{view-write-cert}}$ следует из $(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\mathbf{view}}$ и того, что все экземпляры чтения, предшествующие завершённому экземпляру барьера, также являются завершёнными по свойствам модели ARMv8 POP. □

# Приложение Г. Представление программ

## Г.1 Помеченная система переходов

Первым шагом для построения по программе помеченной системы переходов является генерация всех возможных *путей* ($Path$) исполнения, каждый из которых является списком меток:

$$\begin{aligned} Path &::= \mathit{List}\ \mathit{Label} \\ \mathit{Label} &::= \mathtt{R}(\ell,v) \mid \mathtt{W}(\ell,v) \mid \mathtt{F}(\mathit{fmod}) \mid \varepsilon \end{aligned}$$

Далее по множеству путей строится конечный автомат, принимающий упомянутые пути.

Функция построения путей cmds-to-lbls использует вспомогательную функцию inst-to-lbl, которая принимает список инструкций потока $ilist$, указатель на текущую инструкцию $pos$ и состояние локальных переменных $st$:

cmds-to-lbls : $cmds \to \mathbb{P}(Path)$
cmds-to-lbls $ilist$ = inst-to-lbl $ilist\ 0\ (\lambda \mathit{reg}.0)$

inst-to-lbl : $cmds \to \mathbb{N} \to (\mathit{Reg} \to \mathbb{N}) \to \mathbb{P}(Path)$
inst-to-lbl $ilist\ pos\ st \triangleq$
  if $pos < 0 \mid\mid pos > length(ilist)$ then $\{[]\}$
  else
    **match** $ilist[pos]$ **with**
    |“nop” $\to$ inst-to-lbl $ilist\ (pos+1)\ st$
    |“*reg* := [*expr*]” $\to$
      $\{\mathtt{R}([\![\mathit{expr}]\!]^{st},v) : l \mid \forall v \in \mathit{Val}, l \in$ inst-to-lbl $ilist\ (pos+1)\ st[\mathit{reg} \mapsto v]\}$
    |“*reg* := *expr*” $\to$ inst-to-lbl $(pos+1)\ st[\mathit{reg} \mapsto [\![\mathit{expr}]\!]^{st}]$
    |“[$\mathit{expr}_0$] := $\mathit{expr}_1$” $\to$
      $\{\mathtt{W}([\![\mathit{expr}_0]\!]^{st},[\![\mathit{expr}_1]\!]^{st}) : l \mid \forall l \in$ inst-to-lbl $ilist\ (pos+1)\ st\}$
    |“fence(*fmod*)” $\to \{\mathtt{F}(\mathit{fmod}) : l \mid \forall l \in$ inst-to-lbl $ilist\ (pos+1)\ st\}$
    |“if *expr* goto $k$” $\to$ **let** $step \triangleq$ if $[\![\mathit{expr}]\!]^{st}$ then $k$ else $1$ **in**
      inst-to-lbl $(pos + step)\ st$

Важно отметить, что каждый поток (в обещающей машине) имеет свою систему переходов, которая представляет программу этого потока и, соответственно, строится с помощью функции cmds-to-lbls.

## Г.2 Предзапуски

Для построения по программе ARM-согласованных исполнений стандартно используется следующая схема [65]. В начале по программе строятся *предзапуски* — графы исполнений, в которых определена только часть нужных отношений между событиями, а именно отношения порядка $\texttt{po}$ и зависимостей по данным $\texttt{data}$, управлению $\texttt{ctrl}$ и адресу $\texttt{addr}$.

$$\text{PreExecs} \triangleq \mathbb{P}(\langle set : \mathbb{P}(E), lab : set \rightharpoonup Label,$$
$$\texttt{po} : \mathbb{P}(set \times set), \texttt{ctrl} : \mathbb{P}(set \times set), \texttt{addr} : \mathbb{P}(set \times set), \texttt{data} : \mathbb{P}(set \times set)\rangle)$$

При этом для каждой инструкции чтения генерируется столько вариантов соответствующего события, сколько существует возможных значений соответствующего типа данных. Далее для каждого предзапуска недетерменированно выбираются отношения $\texttt{mo}$ и $\texttt{rf}$ таким образом, чтобы получалось ARM-согласованное исполнение. Важно отменить, что не для всех предзапусков существуют подходящие $\texttt{mo}$ и $\texttt{rf}$.

Функция построения предзапусков по списку команд потока cmds-to-vertices использует вспомогательную функцию inst-to-vertex, которая принимает список инструкций потока $ilist$, указатель на текущую инструкцию $pos$, состояние локальных переменных $st$ и отношение зависимости локальной переменной от сгенерированного события *dep*. Последний параметр нужен для отношений зависимости.

$$\text{cmds-to-vertices} : cmds \to \text{PreExecs}$$
$$\text{cmds-to-vertices}\ ilist \triangleq \text{inst-to-vertex}\ ilist\ 0\ (\lambda reg.0)\ \varnothing$$

inst-to-vertex : $cmds \to \mathbb{N} \to (Reg \to \mathbb{N}) \to \mathbb{P}(Reg \times E) \to$ PreExecs
inst-to-vertex $ilist\ pos\ st\ dep \triangleq$
  if $pos < 0 \mid\mid pos > length(ilist)$ then $\{\langle \varnothing, \bot, \varnothing, \varnothing, \varnothing, \varnothing \rangle\}$
  else
    **let** $a \triangleq$ fresh-vertex **in**
    **match** $ilist[pos]$ **with**
    | "nop" $\to$ inst-to-vertex $ilist\ (pos+1)\ st\ dep$
    | "$reg\ :=\ [expr]$" $\to$
      **let** $dep' \triangleq (dep \setminus \{reg\} \times E) \cup \{\langle reg, a \rangle\}$ **in**
      **let** $\ell \triangleq [\![expr]\!]^{st}$ **in**
      **let** $\mathtt{addr}' \triangleq codom([\mathrm{regs}(expr)]; dep) \times \{a\}$ **in**
      $\{\langle set \cup \{a\}, lab[a \mapsto \mathtt{R}(\ell, v)], \mathtt{po} \cup \{a\} \times set, \mathtt{ctrl}, \mathtt{addr} \cup \mathtt{addr}', \mathtt{data} \rangle \mid \forall v \in Val,$
        $\langle set, lab, \mathtt{po}, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \in$ inst-to-vertex $ilist\ (pos+1)\ st[reg \mapsto v]\ dep'\}$
    | "$reg\ :=\ expr$" $\to$
      **let** $dep' \triangleq (dep \setminus \{reg\} \times E) \cup \{reg\} \times \mathrm{regs}(expr)\}$ **in**
      inst-to-vertex $ilist\ (pos+1)\ st[reg \mapsto [\![expr]\!]^{st}]\ dep'$
    | "$[expr_0]\ :=\ expr_1$" $\to$
      **let** $\ell \triangleq [\![expr_0]\!]^{st}$ **in**
      **let** $v \triangleq [\![expr_1]\!]^{st}$ **in**
      **let** $\mathtt{addr}' \triangleq codom([\mathrm{regs}(expr_0)]; dep) \times \{a\}$ **in**
      **let** $\mathtt{data}' \triangleq codom([\mathrm{regs}(expr_1)]; dep) \times \{a\}$ **in**
      $\{\langle set \cup \{a\}, lab[a \mapsto \mathtt{W}(\ell, v)], \mathtt{po} \cup \{a\} \times set, \mathtt{ctrl}, \mathtt{addr} \cup \mathtt{addr}', \mathtt{data} \cup \mathtt{data}' \rangle \mid$
        $\langle set, lab, \mathtt{po}, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \in$ inst-to-vertex $ilist\ (pos+1)\ st\ dep\}$
    | "fence($fmod$)" $\to$
      $\{\langle set \cup \{a\}, lab[a \mapsto \mathtt{F}(fmod)], \mathtt{po} \cup \{a\} \times set, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \mid$
        $\langle set, lab, \mathtt{po}, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \in$ inst-to-vertex $ilist\ (pos+1)\ st\ dep\}$
    | "if $expr$ goto $k$" $\to$
      **let** $step \triangleq$ if $[\![expr]\!]^{st}$ then $k$ else $1$ **in**
      **let** $dver \triangleq codom([\mathrm{regs}(expr)]; dep)$ **in**
      $\{\langle set, lab, \mathtt{po}, \mathtt{ctrl} \cup dver \times set, \mathtt{addr}, \mathtt{data} \rangle \mid$
        $\langle set, lab, \mathtt{po}, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \in$ inst-to-vertex $ilist\ (pos + step)\ st\ dep\}$

В определении функции inst-to-vertex используется вспомогательная функция regs, которая по выражению вычисляет множество регистров, от которых это выражение зависит.

$$
\begin{array}{l}
\text{regs} : \mathsf{e} \to \mathbb{P}(\mathit{Reg}) \\
\text{regs}\ e \triangleq \\
\quad \textbf{match}\ e\ \textbf{with} \\
\quad \mid \ell \mid v \to \varnothing \\
\quad \mid \mathit{reg} \to \{\mathit{reg}\} \\
\quad \mid \mathit{uop}\ \mathit{expr} \to \text{regs}\ \mathit{expr} \\
\quad \mid \mathit{bop}\ \mathit{expr}_o\ \mathit{expr}_1 \to (\text{regs}\ \mathit{expr}_0) \cup (\text{regs}\ \mathit{expr}_1)
\end{array}
$$

Как было отмечено выше, функция cmds-to-vertices строит предзапуски только одного потока. Для того, чтобы получить предзапуск всей программы, предзапуски потоков программы должны быть покомпонентно объединены, а также в каждый получившийся предзапуск должны быть добавлены события инициализирующей записи:

$$
\begin{array}{l}
\text{prog-to-vertex} : \text{Prog} \to \text{PreExecs} \\
\text{prog-to-vertex}\ \mathit{Prog} \triangleq \\
\quad \textbf{let}\ E(\mathit{tid}) \triangleq \text{cmds-to-vertices}\ \mathit{Prog}(\mathit{tid})\ \textbf{in} \\
\quad \textbf{let}\ P \triangleq \{\langle \bigcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathit{set}, \bigsqcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathtt{lab}, \bigcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathtt{po}, \\
\qquad \bigcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathtt{ctrl}, \bigcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathtt{addr}, \bigcup_{\mathit{tid} \in \mathit{dom}(\mathit{Prog})} \mathit{pe}_{\mathit{tid}}.\mathtt{data} \rangle \\
\qquad \mid \mathit{tid} \in \mathit{dom}(\mathit{Prog}), \mathit{pe}_{\mathit{tid}} \in E(\mathit{tid})\}\ \textbf{in} \\
\quad \textbf{let}\ \mathtt{E}_{\mathit{init}}, \mathit{lab}_{\mathit{init}} \triangleq \{e_\ell, [e_\ell \mapsto \mathtt{W}(\ell,0)] \mid \ell \in \mathit{Loc}\}\ \textbf{in} \\
\quad \textbf{let}\ \mathtt{lab}' \triangleq \bigsqcup \mathit{lab}_{\mathit{init}}\ \textbf{in} \\
\quad \{\langle \mathit{set} \cup \mathtt{E}_{\mathit{init}}, \mathtt{lab} \sqcup \mathtt{lab}', \mathtt{po} \cup (\mathtt{E}_{\mathit{init}} \times \mathit{set}), \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \\
\qquad \mid \langle \mathit{set}, \mathtt{lab}, \mathtt{po}, \mathtt{ctrl}, \mathtt{addr}, \mathtt{data} \rangle \in P\}
\end{array}
$$

## Г.3 Связь между системой переходов и предзапусками

Для того, чтобы определить связь между системой переходов, на которой строится исполнение потока обещающей модели, и предзапуска потока в модели ARMv8.3, мы вводим отношение $\approx$:

$$
\begin{array}{l}
\approx : \mathit{Label}_{\text{Promise}} \to \mathit{Label}_{\text{ARM}} \to \text{Boolean} \\
\mathit{lbl}_{\text{Promise}} \approx \mathit{lbl}_{\text{ARM}} \triangleq \\
\qquad \textbf{match}\ \mathit{lbl}_{\text{Promise}},\ \mathit{lbl}_{\text{ARM}}\ \textbf{with} \\
\qquad \mid \mathtt{R}(\ell,v), \mathtt{R}(\ell,v) \mid \mathtt{W}(\ell,v), \mathtt{W}(\ell,v) \\
\qquad \mid \mathtt{F}(\mathtt{rel}), \mathtt{F}(\mathtt{sy}) \mid \mathtt{F}(\mathtt{acq}), \mathtt{F}(\mathtt{ld}) \to \text{true} \\
\qquad \mid \_, \_ \to \text{false}
\end{array}
$$

Это отношение, по сути, является отношением компиляции. Так, оно связывает метки переходов чтения и записи с событиями чтения и записи, только если они имеют одинаковые параметры: целевую локацию и значение. Кроме того, отношение $\approx$ связывает метку высвобождающего барьера `F(rel)` с событием полного барьера `F(sy)`, а приобретающего барьера `F(acq)` — с событием `ld`-барьера.

**Теорема 8.** Для любого события из предзапуска, построенного по списку инструкций $cmds$, существует путь в системе переходов, построенной по $cmds$, который имеет эквивалентный событию переход. Кроме того, верно и обратное: для любого перехода некоторого пути в системе переходов, построенной по $cmds$, существует предзапуск $cmds$ и событие в нём, эквивалентное этому переходу.

$$
\begin{aligned}
&\forall cmds.\\
&\quad (\forall \langle set, lbl, \texttt{po}, \_, \_, \_\rangle \in \textsf{cmds-to-vertices}\ cmds.\\
&\qquad \exists path \in \textsf{cmds-to-lbls}\ cmds[\mathit{tid}].\ \forall n \in \mathbb{N}, a \in \text{nth}\ \texttt{po}\ n.\\
&\qquad\quad \exists k \in \mathbb{N}.\ path[n+k] \approx lbl\ a) \wedge\\
&\quad (\forall path \in \textsf{cmds-to-lbls}\ cmds.\\
&\qquad \exists \langle set, lbl, \texttt{po}, \_, \_, \_\rangle \in \textsf{cmds-to-vertices}\ cmds.\ \forall n \in \mathbb{N}.\\
&\qquad\quad path[n] \neq \varepsilon \Rightarrow \exists k \in \mathbb{N}, a \in \text{nth}\ \texttt{po}\ (n-k).\ path[n] \approx lbl\ a).
\end{aligned}
$$

Теорема тривиальным образом обобщается на случай программ, т.е. функций, которые по идентификатору потока возвращают его список инструкций.

В формулировке теоремы 8 присутствует функция nth, которая по бинарному отношению $rel$ и натуральному числу $n$ возвращает множество элементов из области определения $rel$, до которых в $rel$ существует путь длины $n$, но не $n+1$:

$$
\begin{aligned}
&\text{nth} : \{A : Set\} \to \mathbb{P}(A \times A) \to \mathbb{N} \to \mathbb{P}(A)\\
&\text{nth}\ rel\ n \triangleq \mathit{codom}(rel^n) \setminus \mathit{codom}(rel^{n+1}).
\end{aligned}
$$

*Доказательство теоремы 8.* Верно по определению функций cmds-to-lbls и cmds-to-vertices. □

# Приложение Д. Доказательство леммы о шаге симуляции обещающей модели и обхода сценария ARMv8.3

**Лемма 11.** Пусть для некоторых конфигураций обхода $\langle C, I\rangle$ и $\langle C', I'\rangle$ сценария $G$, а также некоторого состояния обещающей машины $\langle \mathcal{TS}, M\rangle$ выполняется $G \vdash \langle C, I\rangle \to_{\text{TC}} \langle C', I'\rangle$ и $\mathcal{I}(C, I, \mathcal{TS}, M)$. Тогда существуют такие $\mathcal{TS}'$ и $M'$, что $\langle \mathcal{TS}, M\rangle \xrightarrow[\text{Promise}]{}{}^{+} \langle \mathcal{TS}', M'\rangle$ и $\mathcal{I}(C', I', \mathcal{TS}', M')$.

*Доказательство.* Существует два варианта: $G \vdash C, I \to_{\text{TC}} \langle C', I'\rangle$ соответствует покрытию или выпуску некоторого события $e \in \texttt{E}$. Введем обозначения $tid \triangleq \texttt{tid}(e)$ и $\langle \sigma, \mathcal{V}, \mathit{promises}\rangle \triangleq \mathcal{TS}(tid)$. Начнем с рассмотрения варианта, когда $G \vdash C, I \to_{\text{TC}} \langle C', I'\rangle$ соответствует выпуску события записи $e$.

**Выпуск события** $e$. Из определения $\to_{\text{TC}}$ следует, что $C' = C$, $I' = I \cup \{e\}$ и $e \in \mathsf{Issuable}(G, C, I) \setminus I$. Введем обозначения:

$$\begin{aligned} \ell,\ v,\ \tau &\triangleq \texttt{loc}(e),\ \texttt{val}_{\texttt{w}}(e),\ T(e) \\ \mathit{mview} &\triangleq [\ell@\tau] \sqcup \mathcal{V}.\mathsf{rel} \\ \mathit{msg} &\triangleq \langle \ell : v@\tau, \mathit{mview}\rangle \end{aligned}$$

Мы знаем, что в $M$ нету сообщения для локации $\ell$ с меткой времени $\tau$, т.к. функция $T$ выдает уникальные для одной локации метки времени для событий записи (**correct-tmap**$(G, T)$), и у каждого сообщения из $M$ существует соответствующее ему сообщение в $I$ ($\mathcal{I}_{\textbf{mem2}}(C, I, M)$).

$$\begin{aligned} M',\ \mathit{promises}' &\triangleq M \cup \{\mathit{msg}\},\ \mathit{promises} \cup \{\mathit{msg}\} \\ \mathcal{TS}' &\triangleq \mathcal{TS}[tid \mapsto \langle \sigma, \mathcal{V}, \mathit{promises}'\rangle] \end{aligned}$$

$\mathit{mview} \in M'$, т.к. $[\ell@\tau] \in M'$ и $\mathcal{V}.\mathsf{rel} \in M$. Таким образом $\langle\langle \sigma, \mathcal{V}, \mathit{promises}\rangle, M\rangle \rightsquigarrow_{\mathsf{Promise}\ tid} \langle\langle \sigma, \mathcal{V}, \mathit{promises}'\rangle, M'\rangle$ выполняется. Осталось проверить, что $\mathcal{I}(C, I', \mathcal{TS}', M')$ выполняется.

- $\mathcal{I}_{\textbf{mem1}}(C, I', \mathcal{TS}', M') \wedge \mathcal{I}_{\textbf{mem2}}(C, I', M')$:
  Единственным нетривиальным утверждением, которое нужно проверить, является $\mathit{mview} \leqslant$ dom-view(msg-rel; $[e]$). По определению, $\mathit{mview} = [\ell@\tau] \sqcup \mathcal{V}.\mathsf{rel}\ [\ell@\tau] = [\texttt{loc}(e)@T(e)] \leqslant$ dom-view(msg-rel; $[e]$), т.к. $\langle e, e\rangle \in$ msg-rel. Из $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$ следует $\mathcal{V}.\mathsf{rel} \leqslant$ dom-view(rel-rel; $[e']$), где $e' \in \mathsf{Next}(G, C)$ и $\texttt{tid}(e') = \texttt{tid}(e)$. Т.к. $e \notin C$ и по определению Next, $\langle e', e\rangle \in \texttt{po}^{?}$. Из определения rel-rel и msg-rel следует

rel-rel; $[e']$; $\mathtt{po}^?$; $[e] \subseteq$ msg-rel; $[e]$. Утверждение выполняется, т.к. $\mathcal{V}.\mathsf{rel} \leqslant$ dom-view(rel-rel; $\mathtt{po}$; $[e']$).

- $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}')$: выполняется по $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ и определениям $mview$ и $\mathcal{TS}'$.
- $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS}')$: т.к. $\forall tid.\ \mathcal{TS}'(tid).\mathcal{V} = \mathcal{TS}(tid).\mathcal{V}$ и $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS}')$, инвариант выполняется.
- $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS}')$: следует из того, что $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ выполняется и, для любого $tid$, $\mathcal{TS}(tid).\sigma = \mathcal{TS}'(tid).\sigma$.

**Покрытие события** $e$**.** В этом случае $C' = C \cup \{e\}$, $I' = I$. Т.к. $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ выполняется, существуют такие $t$ и $\sigma'$, что $t \approx \mathtt{lab}(e)$. Рассмотрим варианты $e$.

- $e \in \mathtt{F}(\mathtt{ld})$. В этом случае label$(t) = F\ \mathtt{acq}$ по $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$.

$$\begin{aligned} \mathcal{V}' &\triangleq \langle \mathcal{V}.\mathsf{acq}, \mathcal{V}.\mathsf{acq}, \mathcal{V}.\mathsf{rel} \rangle \\ \mathcal{TS}', M' &\triangleq \mathcal{TS}[tid \mapsto \langle \sigma', \mathcal{V}', \mathit{promises} \rangle], M \end{aligned}$$

Проверим, что $\mathcal{I}(C', I, \mathcal{TS}', M)$ выполняется.

- $\mathcal{I}_{\textbf{mem1}}(C', I, \mathcal{TS}', M) \wedge \mathcal{I}_{\textbf{mem2}}(C', I, M)$: выполняется, т.к. $e \notin \mathtt{W}$ и $\mathcal{I}(C, I, \mathcal{TS}, M)$ выполняется.
- $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}')$: выполняется, т.к. $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ выполняется и $\mathcal{TS}'(tid).\{\mathcal{V}.\mathsf{rel}, \mathit{promises}\} = \mathcal{TS}(tid).\{\mathcal{V}.\mathsf{rel}, \mathit{promises}\}$.
- $\mathcal{I}_{\textbf{view}}(C', \mathcal{TS}')$:

$$\begin{aligned} \forall e' \in \mathsf{Next}(G, C').\ \mathtt{let}\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle \triangleq \mathcal{TS}'(\mathtt{tid}(e')).\mathcal{V}\ \mathtt{in} \\ \mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge \\ \mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge \\ \mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e']). \end{aligned}$$

Зафиксируем $e'$. Если $\mathtt{tid}(e') \neq tid$, то утверждение следует из $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$. Предположим, что $\mathtt{tid}(e') = tid$. Тогда $\mathtt{po}|_{\text{imm}}(e, e')$, т.к. $e \in \mathsf{Next}(G, C)$. Мы знаем, что $\mathcal{V}.\mathsf{acq} \leqslant$ dom-view(acq-rel; $\mathtt{po}$; $[e]$), т.к. $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$ выполняется. Нам нужно показать, что

$$\begin{aligned} \mathcal{V}.\mathsf{acq} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge \\ \mathcal{V}.\mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge \\ \mathcal{V}.\mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e']). \end{aligned}$$

Т.к. $\mathit{dom}$(acq-rel; $[e]$) $\subseteq$ $\mathit{dom}$(acq-rel; $[e']$) и acq-rel; $[e]$; $\mathtt{po}$; $[e'] \subseteq$ cur-rel; $[e']$, утверждение выполняется.

- $\mathcal{I}_{\textbf{state}}(C', \mathcal{TS}')$: очевидно следует из $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ и определений $C', \mathcal{TS}'$.

– $e \in \mathtt{F}(\mathtt{sy})$. В этом случае label$(t) = F$ `rel` по $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$.

$$\begin{aligned}\mathcal{V}' &\triangleq \langle \mathsf{cur}, \mathsf{acq}, \mathsf{cur}\rangle\\ \mathcal{TS}' &\triangleq \mathcal{TS}[tid \mapsto \langle \sigma', \mathcal{V}', \mathit{promises}\rangle]\\ M' &\triangleq M\end{aligned}$$

Не существует $w \in I$ такого, что $\mathtt{po}(e, w)$. Иначе не это противоречило бы $w \in \mathsf{Issuable}(G, C, I)$ по лемме 9 Из этого следует, что не существует $w \in I \setminus C$ такого, что $\mathtt{tid}(w) = tid$, и *promises* $= \varnothing$ по $\mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M)$. Проверим $\mathcal{I}(C', I, \mathcal{TS}', M)$.

- $\mathcal{I}_{\textbf{mem1}}(C', I, \mathcal{TS}', M) \wedge \mathcal{I}_{\textbf{mem2}}(C', I, M)$: выполняется, т.к. $e \notin \mathtt{W}$ и $\mathcal{I}(C, I, \mathcal{TS}, M)$ выполняется.
- $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}')$:
  Зафиксируем поток $tid'$. Если $tid' \neq tid$, то утверждение выполняется по $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$. Если $tid' = tid$, то утверждение выполняется, т.к. $\mathcal{TS}'(tid).\mathit{promises} = \mathcal{TS}(tid).\mathit{promises} = \varnothing$.
- $\mathcal{I}_{\textbf{view}}(C', \mathcal{TS}')$:

$$\begin{aligned}\forall e' \in \mathsf{Next}(G, C').\ \mathtt{let}\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel}\rangle \triangleq \mathcal{TS}'(\mathtt{tid}(e')).\mathcal{V}\ \mathtt{in}\\ \mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge\\ \mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge\\ \mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e']).\end{aligned}$$

  Зафиксируем $e'$. Если $\mathtt{tid}(e') \neq tid$, то утверждение следует из $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$. Если $\mathtt{tid}(e') = tid$, то $\mathtt{po}|_{\text{imm}}(e, e')$, т.к. $e \in \mathsf{Next}(G, C)$. Нам нужно показать, что

$$\begin{aligned}\mathcal{V}.\mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge\\ \mathcal{V}.\mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge\\ \mathcal{V}.\mathsf{cur} \leqslant \text{dom-view}(\text{rel-rel}; [e']).\end{aligned}$$

  Т.к. $\mathit{dom}(\text{cur-rel}; [e]) \subseteq \mathit{dom}(\text{cur-rel}; [e'])$ и $\text{rel-rel}; [e]; \mathtt{po}; [e'] \subseteq \text{cur-rel}; [e']$, утверждение выполняется.
- $\mathcal{I}_{\textbf{state}}(C', \mathcal{TS}')$: очевидно следует из $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ и определений $C', \mathcal{TS}'$.

– $e \in \mathtt{R}$. В этом случае $\mathrm{label}(t) = R\ (\ell, v)$ по $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$.

$$\begin{aligned} \ell, v &\triangleq \mathtt{loc}(e), \mathtt{val_r}(e) \\ \sigma, \mathcal{V}, \mathit{promises} &\triangleq \mathcal{TS}(tid) \end{aligned}$$

Т.к. $e \in \mathtt{R} \cap \mathsf{Coverable}(G, C, I)$ из определения $\rightarrow_{\mathrm{TC}}$, существует событие записи $w \in I \cap \mathit{dom}(\mathtt{rf}; [e])$. По $\mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M)$ существует фронт $view$ такой, что $msg \triangleq \langle \ell : v@T(w), view \rangle \in M$. $\mathcal{V}.\mathsf{cur} \leqslant T(w)$ по лемме 15.

$$\begin{aligned} \mathcal{V}' &\triangleq \langle \mathcal{V}.\mathsf{cur} \sqcup [\ell@T(w)], \mathcal{V}.\mathsf{acq} \sqcup view, \mathcal{V}.\mathsf{rel} \rangle \\ \mathcal{TS}' &\triangleq \mathcal{TS}[tid \mapsto \langle \sigma', \mathcal{V}', \mathit{promises} \rangle] \end{aligned}$$

Нужно проверить $\mathcal{I}(C', I, \mathcal{TS}', M)$.

– $\mathcal{I}_{\textbf{mem1}}(C', I, \mathcal{TS}', M) \wedge \mathcal{I}_{\textbf{mem2}}(C', I, M)$: выполняется, т.к. $e \notin \mathtt{W}$ и $\mathcal{I}(C, I, \mathcal{TS}, M)$ выполняется.
– $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}')$: выполняется, т.к. $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ выполняется и $\mathcal{TS}'(tid).\{\mathcal{V}.\mathsf{rel}, \mathit{promises}\} = \mathcal{TS}(tid).\{\mathcal{V}.\mathsf{rel}, \mathit{promises}\}$.
– $\mathcal{I}_{\textbf{view}}(C', \mathcal{TS}')$:

$$\begin{aligned} \forall e' \in \mathsf{Next}(G, C').\ \textbf{let}\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle \triangleq \mathcal{TS}'(\mathtt{tid}(e')).\mathcal{V}\ \textbf{in} \\ \mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge \\ \mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge \\ \mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e']). \end{aligned}$$

Зафиксируем $e'$. Если $\mathtt{tid}(e') \neq tid$, то утверждение следует из $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$. Предположим, что $\mathtt{tid}(e') = tid$. Тогда $\mathtt{po}|_{\mathrm{imm}}(e, e')$, т.к. $e \in \mathsf{Next}(G, C)$. Нам нужно показать, что

$$\begin{aligned} \mathcal{V}.\mathsf{cur} \sqcup [\ell@T(w)] &\leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge \\ \mathcal{V}.\mathsf{acq} \sqcup view &\leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge \\ \mathcal{V}.\mathsf{rel} &\leqslant \text{dom-view}(\text{rel-rel}; [e']). \end{aligned}$$

Т.к. $\langle w, e' \rangle \in$ cur-rel, $[\ell@T(w)] \leqslant$ dom-view(cur-rel; $[e']$). Из $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ следует, что $view = [\ell@T(w)] \sqcup \mathcal{TS}(\mathtt{tid}(w)).\mathcal{V}.\mathsf{rel}$. Т.к. $\langle w, e' \rangle \in$ acq-rel, $[\ell@T(w)] \leqslant$ dom-view(acq-rel; $[e']$). Т.к. $\mathcal{TS}(\mathtt{tid}(w)).\mathcal{V}.\mathsf{rel} \leqslant$ dom-view(rel-rel; $[w]$) и $\mathit{dom}$(rel-rel; $[w]$) $\subseteq$ $\mathit{dom}$(acq-rel; $[e']$), утверждение выполняется.
– $\mathcal{I}_{\textbf{state}}(C', \mathcal{TS}')$: очевидно следует из $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ и определений $C', \mathcal{TS}'$.

– $e \in \mathtt{W}$. В этом случае label$(t) = W\ (\ell, v)$ по $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$.

$$\begin{aligned}\ell, v, \tau &\triangleq \mathtt{loc}(e), \mathtt{val_w}(e), T(e)\\ \langle \sigma, \mathcal{V}, \mathit{promises}\rangle &\triangleq \mathcal{TS}(\mathit{tid})\\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel}\rangle &\triangleq \mathcal{V}.\end{aligned}$$

$e \in I$, т.е. $e \in \mathtt{W} \cap \mathsf{Coverable}(G, C, I)$. Из $\mathcal{I}_{\textbf{mem1}}(C, I, \mathcal{TS}, M)$ следует, что существует $\mathit{view}$ такое, что $\mathit{msg} \triangleq \langle \ell : v@\tau, \mathit{view}\rangle \in M$. $\mathsf{cur}(\ell) < \tau$ следует по лемме 15.

$$\begin{aligned}\mathsf{cur}', \mathsf{acq}' &\triangleq \mathsf{cur} \sqcup [\ell@\tau], \mathsf{acq} \sqcup [\ell@\tau]\\ \mathcal{V}' &\triangleq \langle \mathsf{cur}', \mathsf{acq}', \mathsf{rel}\rangle\\ \mathcal{TS}' &\triangleq \mathcal{TS}[\mathit{tid} \mapsto \langle \sigma', \mathcal{V}', \mathit{promises} \setminus \mathit{msg}\rangle]\end{aligned}$$

$\mathit{view} = \mathsf{rel}$ по $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$.

Нужно проверить $\mathcal{I}(C', I, \mathcal{TS}', M)$.

– $\mathcal{I}_{\textbf{mem1}}(C', I, \mathcal{TS}', M) \wedge \mathcal{I}_{\textbf{mem2}}(C', I, M)$: выполняется, т.к. мы добавили $e$ во множество покрытых событий и убрали $\mathit{msg}$ из множества обещанных сообщений потока $\mathit{tid}$.
– $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS}')$: выполняется, т.к. $\mathcal{I}_{\textbf{view-rel}}(\mathcal{TS})$ выполняется, $\mathcal{TS}'(\mathit{tid}).\mathcal{V}.\mathsf{rel} = \mathcal{TS}(\mathit{tid}).\mathcal{V}.\mathsf{rel}$ и $\mathcal{TS}'(\mathit{tid}).\mathit{promises} \subset \mathcal{TS}(\mathit{tid}).\mathit{promises}$.
– $\mathcal{I}_{\textbf{view}}(C', \mathcal{TS}')$:

$$\begin{aligned}&\forall e' \in \mathsf{Next}(G, C').\ \mathtt{let}\ \langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel}\rangle \triangleq \mathcal{TS}'(\mathtt{tid}(e')).\mathcal{V}\ \mathtt{in}\\ &\quad \mathsf{cur} \leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge\\ &\quad \mathsf{acq} \leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge\\ &\quad \mathsf{rel} \leqslant \text{dom-view}(\text{rel-rel}; [e']).\end{aligned}$$

Зафиксируем $e'$. Если $\mathtt{tid}(e') \neq \mathit{tid}$, то утверждение следует из $\mathcal{I}_{\textbf{view}}(C, \mathcal{TS})$. Предположим, что $\mathtt{tid}(e') = \mathit{tid}$. Тогда $\mathtt{po}|_{\text{imm}}(e, e')$, т.к. $e \in \mathsf{Next}(G, C)$. Нам нужно показать, что

$$\begin{aligned}\mathcal{V}.\mathsf{cur} \sqcup [\ell@\tau] &\leqslant \text{dom-view}(\text{cur-rel}; [e']) \wedge\\ \mathcal{V}.\mathsf{acq} \sqcup [\ell@\tau] &\leqslant \text{dom-view}(\text{acq-rel}; [e']) \wedge\\ \mathcal{V}.\mathsf{rel} &\leqslant \text{dom-view}(\text{rel-rel}; [e']).\end{aligned}$$

Т.к. $\langle w, e'\rangle \in \text{cur-rel} \subseteq \text{acq-rel}$, $[\ell@\tau] \leqslant \text{dom-view}(\text{cur-rel}; [e']) \leqslant \text{dom-view}(\text{acq-rel}; [e'])$.

– $\mathcal{I}_{\textbf{state}}(C', \mathcal{TS}')$: очевидно следует из $\mathcal{I}_{\textbf{state}}(C, \mathcal{TS})$ и определений $C', \mathcal{TS}'$.

□

**Лемма 15.** Для любой локации $\ell$ выполняется $[\mathtt{W}_\ell]; \text{cur-rel}; [\mathtt{W}_\ell] \subseteq \mathtt{mo}$.

*Доказательство.* Зафиксируем $\langle w, w' \rangle \in [\mathtt{W}_\ell]; \text{cur-rel}; [\mathtt{W}_\ell]$. Тогда по определению $\mathtt{mo}$, либо $\mathtt{mo}(w, w')$, либо $\mathtt{mo}(w', w)$. Если выполняется первое, то выполняется утверждение. Пусть выполняется второе.

$$
\begin{aligned}
&[\mathtt{W}_\ell]; \text{cur-rel}; [\mathtt{W}_\ell] = \\
&[\mathtt{W}_\ell]; \mathtt{rf}^?; (\mathtt{po} \cup \mathtt{sw})^+; [\mathtt{W}_\ell] = \\
&\quad \text{(по транзитивности } \mathtt{po} \text{ и определению } \mathtt{sw}) \\
&[\mathtt{W}_\ell]; \mathtt{rf}^?; \mathtt{po}; (\mathtt{sw}; \mathtt{po})^*; [\mathtt{W}_\ell] = \\
&[\mathtt{W}_\ell]; \mathtt{rf}^?; \mathtt{po}; [\mathtt{W}_\ell] \cup \\
&[\mathtt{W}_\ell]; \mathtt{rfe}^?; \mathtt{po}; (\mathtt{sw}; \mathtt{po})^+; [\mathtt{W}_\ell] \cup
\end{aligned}
$$

$\langle w, w' \rangle \notin [\mathtt{W}_\ell]; \mathtt{rf}^?; \mathtt{po}; [\mathtt{W}_\ell]$, т.к. $\langle w', w \rangle \in \mathtt{mo}$ и выполняется INTERNAL.

Введем вспомогательное отношение $\mathtt{eord} \triangleq (\mathtt{obs} \cup \mathtt{dob} \cup \mathtt{bob})^+$, которое антирефлексивно по определению ARM-согласованности (EXTERNAL). Из определений отношений следует, что $\mathtt{po}^?; \mathtt{sw}; \mathtt{po} \subseteq \mathtt{bob}^?; \mathtt{bob} \cup \mathtt{bob}^?; \mathtt{bob}; \mathtt{rfe}; \mathtt{bob} \subseteq \mathtt{eord}$. По транзитивности $\mathtt{eord}$, $\langle w, w' \rangle \in \mathtt{eord}$.

Мы знаем, что $\langle w', w \rangle \in \mathtt{mo}$. Существует два варианта: $\langle w', w \rangle \in \mathtt{moe}$ или $\langle w', w \rangle \in \mathtt{moi}$. Первый вариант противоречит ацикличности $\mathtt{eord}$, т.к. $\mathtt{moe} \subseteq \mathtt{obs}$. Опровергнем второй вариант, показав, что $(\mathtt{eord} \cup \mathtt{moi})^+ = (\mathtt{obs} \cup \mathtt{dob} \cup \mathtt{bob} \cup \mathtt{moi})^+$ антирефлексивно.

Предположим обратное и рассмотрим цикл минимальной длины, в котором точно есть $\mathtt{moi}$ по антирефлексивности $\mathtt{eord}$. Рассмотрим предыдущее ребро цикла, которое мы обозначим $r$, и покажем, что во всех случаях цикл можно укоротить.

– Если $r \in \mathtt{moi}$, то цикл может быть укорочен, т.к. $\mathtt{moi}; \mathtt{moi} \subseteq \mathtt{moi}$.
– Если $r \in \mathtt{obs} = \mathtt{rfe} \cup \mathtt{rbe} \cup \mathtt{moe}$, то нужно рассмотреть три варианта. Поскольку $\mathtt{rfe}; \mathtt{moi} = \varnothing$, то $r$ не может быть из отношения $\mathtt{rfe}$. В двух других случаях цикл может быть укорочен, т.к. $\mathtt{rbe}; \mathtt{moi} \subseteq \mathtt{rbe}$, и $\mathtt{moe}; \mathtt{moi} \subseteq \mathtt{moe}$.
– Если $r \in \mathtt{bob}$, то цикл может быть укорочен, поскольку $\mathtt{po}; \mathtt{moi} \subseteq \mathtt{po}$.
– Если $r \in \mathtt{dob}$, то цикл может быть укорочен:

- $\mathtt{addr};\mathtt{po}^{?};\mathtt{moi} \subseteq \mathtt{dob}$,
- $\mathtt{data};\mathtt{moi} \subseteq \mathtt{dob}$,
- $(\mathtt{addr} \cup \mathtt{data});\mathtt{rfi};\mathtt{moi} = \varnothing$,
- $(\mathtt{ctrl} \cup \mathtt{data});[\mathtt{W}];\mathtt{moi}^{?};\mathtt{moi} \subseteq \mathtt{dob}$.

□