Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Кафедра системного программирования

Намаконов Егор Сергеевич

Компиляция модели памяти OCaml в Power

Магистерская диссертация

Научный руководитель:
д.ф.-м.н. Кознов Д. В.

Консультант:
к.ф.-м.н. Подкопаев А.В.

Рецензент:
к.ф.-м.н. Березун Д. А.

Санкт-Петербург
2020

ST. PETERSBURG UNIVERSITY
Software and Administration of Information Systems

Software Engineering

Egor Namakonov

Compilation of OCaml memory model to Power

Graduation Thesis

Scientific supervisor:
D. Sc. Dmitri Koznov

Consultant:
Ph.D. Anton Podkopaev

Reviewer:
Ph.D. Daniil Berezun

St. Petersburg
2020

Оглавление

1. Введение

Результат исполнения многопоточной программы, как правило, является недетерминированным. Конкретное множество допустимых результатов многопоточной программы определяется *моделью памяти* языка программирования. Наиболее известной является *модель последовательной согласованности* (sequential consistency, SC [11]). Она предполагает, что любой результат исполнения программы может быть получен путём попеременного исполнения инструкций отдельных потоков согласно программному порядку в них. Однако из-за оптимизаций, выполняемых современными компиляторами и процессорами, могут наблюдаться сценарии поведения, невозможные в такой модели. Так, на архитектуре x86 чтение по адресу в памяти может вернуть не самое последнее записанное значение, так как операция записи может быть буферизована.

Отказ от подобных оптимизаций нежелателен, поэтому современные модели памяти допускают некоторые сценарии поведения, невозможные в модели SC. Такие модели памяти называются *слабыми*. Например, слабыми являются модели памяти языков C++ [2], JavaScript [22] и Java [13], а также архитектур Power [1], x86 [16] и ARM [19].

Модель памяти OCaml [4] (далее — `OCamlMM`) отличается свойством т.н. *локальной свободы от гонок по данным* (local data race freedom). Именно, гарантируется, что выполнение конфликтующих обращений по выбранному адресу в памяти (т.е. ситуация гонки по данным) не влияет на обращения к другим адресам, а также на последующие обращения по тому же адресу. Благодаря этому даже при возникновении гонки по данным в некоторый момент исполнения следующие участки программы будут исполнены согласно модели SC.

Чтобы гарантировать выполнение этого свойства, при компиляции нужно запретить некоторые оптимизации в зависимости от целевой архитектуры. Для этого может понадобиться, например, добавить в ассемблерный код инструкции-*барьеры*, которые запрещают нежелательные оптимизации на уровне процессора. Набор таких правил, по-

крывающий все возможные типы инструкций, называется *схемой компиляции*. Схема компиляции должна быть *корректной* — при исполнении любой программы, полученной при компиляции согласно этой схеме, должно наблюдаться только сценарии поведения, разрешённые OCamlMM для исходной программы.

Авторы OCamlMM разработали схемы компиляции OCamlMM в модели x86-TSO и ARMv8 [4] и доказали их корректность. При этом отсутствует схема компиляции в модель архитектуры Power. А между тем данная архитектура часто используется в современном серверном оборудовании [7]. Задача построения такой схемы осложнена тем, что модель Power, в отличие от моделей x86-TSO, ARMv8 и OCamlMM, не обладает т.н. свойством *multicopy atomicity*. Такое свойство означает, что записанные в память значения становятся доступны всем потокам в одном и том же порядке [19]. Из-за отсутствия этого свойства корректная схема компиляции OCamlMM в Power должна расставлять барьеры в результирующей программе так, чтобы запретить нежелательные сценарии поведения.

В рамках данной работы была поставлена задача разработать схему компиляции OCamlMM в модель Power и доказать её корректность. Для этого было решено использовать промежуточную модель памяти (Intermediate Memory Model, далее — IMM) [18], для которой уже доказана корректность компиляции в модель Power. Использование IMM как промежуточного этапа компиляции позволяет разбить доказательство корректности на два, которые впоследствии можно использовать в других доказательствах. Таким образом, построение схемы компиляции OCamlMM в IMM даёт схемы компиляции OCamlMM не только в Power, но и другие архитектуры, в которые компилируется IMM (на данный момент — x86 и ARM).

2. Постановка задачи

Целью данной работы является доказательство корректности компиляции модели памяти OCaml (`OCamlMM`) в модель памяти Power [1].

В работе были поставлены следующие задачи:

- построение схемы компиляции `OCamlMM` в `IMM` (для которой корректность компиляции в Power уже доказана);
- доказательство корректности полученной схемы;
- формализация доказательства в системе интерактивного доказательства теорем Coq [20].

3. Обзор

В этом разделе приводится пример слабого поведения программы и объясняются его причины. Затем на примерах рассматривается понятие корректности компиляции для моделей памяти. Далее формально описывается декларативный способ задания модели памяти [1], основанный на понятии графов исполнения. Наконец, формально описываются модели памяти `OCamlMM` и `IMM`, используемые далее в работе.

3.1. Пример исполнения в слабой модели памяти

Рассмотрим программу, представленную на рис. 1. Здесь и далее используется упрощённый синтаксис программ: x и y обозначают адреса в памяти, a и b — локальные переменные (регистры), `rlx` – режим доступа (это понятие будет рассмотрено ниже). Сверху указаны изначальные значения в памяти. В комментариях указаны наблюдаемые при чтении значения. Согласно модели SC, в зависимости от порядка исполнения инструкций в a и b могут быть записаны значения $(1, 1)$, $(1, 0)$ или $(0, 1)$. Однако после компиляции C++-аналога этой программы с помощью компилятора gcc и исполнения на архитектуре x86 в переменные a и b могут быть записаны нули, что не допускается моделью SC. У такого сценария поведения могут быть две причины. Во-первых, gcc может поменять местами обращения по разным адресам во время компиляции. Во-вторых, при исполнении на x86 возможна буферизация записи: в целях оптимизации обращений к памяти запись может быть отложена.

$$x = 0,\ y = 0$$

$[x]^{\texttt{rlx}} := 1;$	$[y]^{\texttt{rlx}} := 1;$
$a := [y]^{\texttt{rlx}}; //0$	$b := [x]^{\texttt{rlx}}; //0$

Рис. 1: Пример программы и её исполнения при буферизации записи

Новые сценарии поведения программы, возникающие в результате оптимизаций, могут быть некорректными с точки зрения требова-

ний к программе. Поэтому слабая модель памяти должна предоставлять возможность отменить оптимизации для отдельных инструкций. Для этого используются *режимы доступа*, различные по степени строгости. При компиляции инструкций с более строгими режимами компилятор может отменить некоторые оптимизации, а также добавить в результирующую программу инструкции-барьеры.

Так, в программе на рис. 1 был использован режим доступа `rlx`, который не ограничивает оптимизации соответствующих инструкций. Модель памяти C++ гарантирует, что если в этой программе для всех инструкций установить режим доступа `sc` вместо `rlx`, то поведение полученной программы будет согласовано с моделью SC. Это справедливо в силу того, что такие обращения будут скомпилированы с использованием инструкции MFENCE [14] — барьера памяти, запрещающего перестановки SC инструкций.

3.2. Проблема корректности компиляции на примерах

Модели `OCamlMM` и `IMM` определены декларативно [1]. Это означает, что каждое возможное поведение программы задаётся в виде *графа исполнения*, а семантика программы определяется как множество графов, удовлетворяющих некоторому условию. Пример программы и одного из графов её исполнения приведён на рис. 2.

Вершины графа соответствуют событиям — операциям над разделяемой памятью, которые производятся при выполнении инструкций программы. Так, событие $\mathtt{W}^{\mathtt{at}}(x, 1)$ соответствует записи по адресу x значения 1 в режиме `at`; другими типами событий являются чтение и барьер памяти (обозначаются `R` и `F` соответственно). Кроме того, в графе выделяются инициализирующие события, которые соответствуют инициализирующей записи нулей в память. На рис. 2 все они для краткости обозначены множеством `Init`; далее в графах мы будем опускать эти события, если это не будет важно для рассуждений. Заметим, что содержимое локальных переменных потока не отражается в гра-

фе в явном виде, т.к. взаимодействие между потоками производится только через разделяемую память.

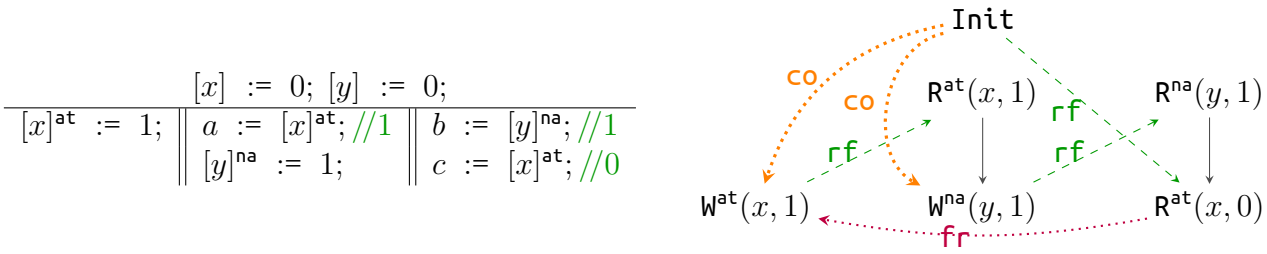

Рис. 2: Пример программы и сценария её исполнения, не согласованного в OCamlMM

Рёбра графа задают бинарные отношения между событиями. В данном графе есть четыре различных отношения: рёбра po соответствуют программному порядку инструкций, rf — чтению записанного ранее значения, co — порядку выполнения записи по одному адресу, fr — чтению до указанного события записи. Отношения po и co являются транзитивными, поэтому для их задания достаточно указывать только непосредственные рёбра. Кроме того, для краткости будем опускать подпись po рядом с соответствующими рёбрами.

Согласованными (допустимыми моделью) называются те сценарии исполнения программы, графы которых удовлетворяют некоторому предикату, заданному моделью. В частности, предикат согласованности OCamlMM требует, чтобы в графе не было циклов, состоящих только из рёбер co и fr, проходящих между вершинами с меткой at, а также рёбер po и rf. Это условие формализует свойство multicopy atomicity, описанное выше.

Граф исполнения на рис. 2 не является согласованным по OCamlMM. Действительно, этот сценарий исполнения нарушает свойство multicopy atomicity: второй поток читает записанное в x значение 1 до записи 1 в y, однако третий поток читает старое значение 0 из x после чтения 1 из y. Соответствующий граф исполнения не удовлетворяет предикату согласованности OCamlMM, так как между вершинами есть цикл, подходящий под описание выше. Таким образом, в OCamlMM после исполнения программы на рис. 2 переменные a, b и c не могут содержать значения 1, 1 и 0 соответственно.

Условие корректности компиляции требует, чтобы сценарии поведения, запрещённые для исходной программы в OCamlMM, также были запрещены для скомпилированной программы в IMM. Для декларативных моделей памяти это означает, что из несогласованности графа исполнения в OCamlMM должна следовать несогласованность соответствующего ему графа исполнения в IMM. При этом, как будет показано далее, можно рассматривать вопрос согласованности по OCamlMM только для графа исполнения скомпилированной программы.

На рис. 3 приведён результат компиляции программы на рис. 2 согласно тривиальной схеме компиляции. Такая схема лишь заменяет режимы инструкций на их аналоги в IMM: `na` заменяется на `rlx`, а `at` — на `sc`; дополнительных инструкций не вводится. Соответственно, граф исполнения на рис. 3 отличается от графа на рис. 2 только метками вершин, и в нём сохраняется цикл того же вида. IMM не гарантирует свойство multicopy atomicity, и потому предикат её согласованности не требует отсутствия таких циклов. Поэтому граф на рис. 3 согласован, что делает соответствующий ему сценарий поведения разрешается IMM, в отличие от OCamlMM. Поэтому тривиальная схема компиляции не является корректной.

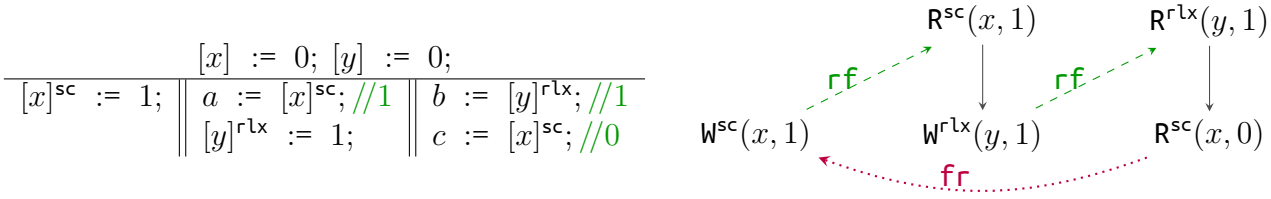

Рис. 3: Результат компиляции программы на рис. 2 с использованием тривиальной схемы компиляции и согласованный по IMM граф его исполнения

На рис. 4 приведена программа, полученная в результате компиляции программы на рис. 2 согласно схеме компиляции, приведённой в разделе 4. В результате компиляции в ней появляются инструкции барьеров памяти, запрещающие некоторые оптимизации процессора и компилятора. С этими барьерами граф исполнения перестаёт быть согласованным: из рёбер `rf` между событиями с меткой `sc`, а также `fr`, `po` и окружённого барьерами `rf` образуется цикл, запрещённый в IMM.

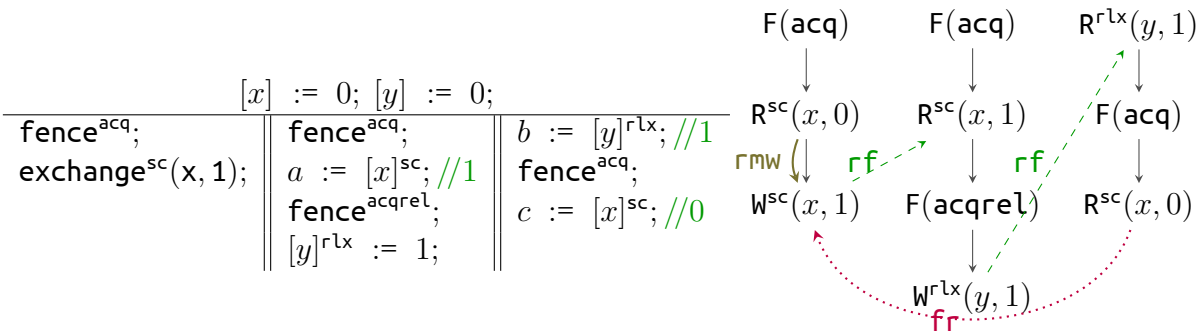

Рис. 4: Результат компиляции программы на рис. 2 с использованием схемы компиляции из раздела 4 и его граф исполнения, не согласованный в IMM

Таким образом, для доказательства корректности компиляции необходимо доказать следующую теорему[1].

***Теорема* 3.1.** *Пусть* $PO = ||_{\tau \in \mathsf{Tid}} PO_\tau$ *и* $PI = ||_{\tau \in \mathsf{Tid}} PI_\tau$ *— программы для моделей* OCamlMM *и* IMM *соответственно, причём для любого* $\tau \in \mathsf{Tid}$ *подпрограмма* PI_τ *получена компиляцией* PO_τ *с помощью схемы компиляции из таб. 1. Пусть* G_I *— согласованный по* IMM *граф исполнения* PI*. Тогда существует* G_O *— согласованный по* OCamlMM *граф исполнения* PO*, соответствующий* G_I*.*

Ключевой идеей доказательства является то, что согласованность по OCamlMM можно рассматривать для обоих графов исполнения. Для доказательства теоремы достаточно показать, что G_I является согласованным по OCamlMM. Из этого следует согласованность G_O по OCamlMM, так как он фактически является подграфом G_I , а условия согласованности по OCamlMM таковы, что выполняются для подграфов. Условие согласованности по OCamlMM состоит в иррефлексивности одного отношения и ацикличности другого. Для каждого из этих отношений доказывается включение в такое отношение, для которого соответствующее условие выполняется в согласованном по IMM графе.

[1]Формальное понятие соответствие графов вводится в разделе 4.

3.3. Графы исполнения

В описаниях декларативных моделей памяти мы будем использовать следующие обозначения отношений между вершинами. Для бинарного отношения R обозначения $R^?$, R^+, R^* соответствуют его рефлексивному, транзитивному и транзитивно-рефлексивному замыканиям соответственно. Обратное отношение записывается как R^{-1}. Левая композиция отношений R_1 и R_2 записывается следующим образом:

$R_1; R_2 \triangleq \{x, y | \exists z.(x, z) \in R_1 \wedge (z, y) \in R_2\}$.

Непосредственные рёбра R обозначаются как $R|_{\text{imm}} \triangleq R \backslash R; R$. Тождественное отношение на множестве A обозначается как $[A]$; в частности, $[A]; R; [B] = R \cap (A \times B)$.

В данном разделе описываются графы исполнения наиболее общего вида, без привязки к конкретным моделям памяти или языкам.

Считаем, что анализируемая программа P состоит из последовательных подпрограмм отдельных потоков P_τ: $P = ||_{\tau \in \mathsf{Tid}} P_\tau$, где $||$ — оператор параллельной композиции программ, а Tid — конечное множество идентификаторов потоков.

***Определение* 3.2.** Граф исполнения G *задаётся множеством вершин, бинарными отношениями на вершинах, а также функцией, сопоставляющей вершинами* метки.

Множество вершин, обозначаемое как $G.\mathtt{E}$, делится на инициализирующие события вида `Init` loc и неинициализирующие события вида `ThreadEvent` τ n. Их компонентами являются:

- $loc \in \mathsf{Loc}$ — адрес инициализации, где Loc — конечное множество адресов;
- $\tau \in \mathsf{Tid}$ — номер потока;
- $n \in \mathbb{N}$ — порядковый номер внутри потока.

Функция $G.\mathsf{Lab}$ сопоставляет событиям *метки* вида $(type, loc, mode, val)$. Их компонентами являются:

- $type \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$ — тип операции (чтение, запись, барьер);
- $loc \in \mathsf{Loc}$ — адрес памяти (для барьера не определено);
- $mode$ — один из режимов доступа (например, `rel`), частично упорядоченных отношением “строже чем” ($\sqsubset$); конкретное множество режимов и их порядок определяется моделью памяти;
- $val \in \mathsf{Val}$ — прочитанное/записанное значение (в случае барьера не определено), где Val — множество значений, которые могут храниться в памяти.

Следует отметить, что инициализирующие события вида $\mathtt{Init}\ loc$ обрабатываются особым образом. Именно,

$G.\mathsf{Lab}(\mathtt{Init}\ loc) = (\mathtt{W}, loc, mode_{\mathsf{Init}}, val_{\mathsf{Init}})$, где $mode_{\mathsf{Init}}$ — выбранный режим доступа для инициализирующих событий записей (например, в `IMM` — `rlx`), а val_{Init} — начальное значение в памяти (как правило, 0).

Для множеств событий с определёнными метками вводятся соответствующие обозначения. Например, события с меткой чтения в режиме `acq` или более строгим будем обозначать как $G.\mathtt{R}^{\mathtt{acq}}$ (или просто $\mathtt{R}^{\mathtt{acq}}$, если граф очевиден из контекста).

Рёбра графа представляют собой следующие отношения между событиями:

- программный порядок (program order): $G.\mathtt{po}(x, y) \iff (x \in \mathsf{Init} \wedge y \notin \mathsf{Init}) \vee (x.\tau = y.\tau \wedge x.n < y.n)$;
- порядок согласованности (coherence order): $G.\mathtt{co} = \bigcup_{l \in \mathsf{Loc}} \mathtt{co}_l$, где $\mathtt{co}_l$ — тотальный порядок на событиях записи по адресу l;
- наблюдение записанного значения (“читает-из”, reads from):

  $G.\mathtt{rf} \subseteq \bigcup_{l \in \mathsf{Loc}} G.\mathtt{W}_l \times G.\mathtt{R}_l$, где

  $G.\mathtt{rf}(w, r) \Rightarrow G.\mathsf{Lab}(w).val = G.\mathsf{Lab}(r).val$, $codom(G.\mathtt{rf}) = G.\mathtt{R}$ и

  $G.\mathtt{rf}^{-1}$ является функциональным отношением;

- чтение до указанной записи: $G.\mathtt{fr} = G.\mathtt{rf}^{-1}; G.\mathtt{co}$ (from-read, “читает-до”).

Различные модели памяти могут иметь в графе исполнения и другие отношения. Например, в модели IMM также есть отношение $\mathtt{rmw} \subseteq \bigcup_{l \in \mathsf{Loc}} [G.\mathtt{R}_l]; \mathtt{po}|_{\mathrm{imm}}; [G.\mathtt{W}_l]$, соответствующее паре событий чтения и записи в операции read-modify-write.

Введём понятия сужения графа на поток i: $G_\tau.\mathtt{E} = \{e \in G.\mathtt{E} \mid e.\tau = i\}$, $G_\tau.\mathsf{Lab} = G.\mathsf{Lab}$.

***Определение* 3.3.** Графом исполнения программы P *называется такой граф* G*, что его сужение на любой поток* $\tau \in \mathsf{Tid}$ *является* однопоточным графом исполнения *программы* P_τ.

Соответствие подпрограммы потока и однопоточного графа исполнения определяется средствами операционной семантики, специфичной для модели памяти [4], [18]. Мы не приводим подробностей здесь, скажем лишь, что такая семантика задаёт соответствие между выполнением инструкций языка и изменением графа исполнения. Так, для IMM выполнение инструкции $[x]^{\mathtt{rel}} := 1$ соответствует добавлению в текущий граф вершины с очередным порядковым номером и меткой вида $\mathtt{W}^{\mathtt{rel}}(x, 1)$, а также рёбер, отражающих синтаксические зависимости данного события.

***Определение* 3.4.** *Граф исполнения определяет* сценарий поведения *программы — функцию* $f : \mathsf{Loc} \to \mathsf{Val}$*, отображающую адрес в последнее (согласно порядку* co*) записанное по нему значение.*

Декларативная модель памяти задаётся предикатом согласованности, которому должны удовлетворять графы исполнения программ.

***Определение* 3.5.** *Сценарий поведения программы является* согласованным *по модели памяти* M*, если он задан некоторым графом её исполнения, удовлетворяющим предикату согласованности* M.

3.4. Описание используемых моделей памяти

В данном разделе описываются рассматриваемые модели памяти — `OCamlMM` и `IMM`, — а также их предикаты согласованности.

3.4.1. Модель памяти OCaml (`OCamlMM`)

Модель памяти OCaml (`OCamlMM`) [4] задана эквивалентными операционным и декларативным описаниями. Для доказательства корректности компиляции будет использоваться декларативное описание.

`OCamlMM` поддерживает два режима доступа: неатомарный `na` и атомарный `at` (схожи с `pln` и `sc` в C++). При этом память также разделена на неатомарные и атомарные адреса, и к конкретному адресу можно обратиться только операцией соответствующего режима.

В графе исполнения `OCamlMM` есть только операции чтения и записи, барьеры отсутствуют.

Перед рассмотрением предиката согласованности введём ещё несколько обозначений. Для отношения R в графе исполнения будем обозначать Ri рёбра R, проходящие между вершинами одного потока, а Re — между вершинами разных потоков.

***Определение* 3.6.** *Сценарий исполнения называется* согласованным по `OCamlMM`*, если в соответствующем графе исполнения выполняются следующие аксиомы:*

1) *последовательная согласованность по отдельным адресам (SC per location, coherence): отношение* $\mathtt{hbo}; (\mathtt{co} \cup \mathtt{fr})$ *иррефлексивно, где* $\mathtt{hbo} \triangleq \mathtt{po} \cup [\mathtt{E}^{\mathtt{at}}]; (\mathtt{rf} \cup \mathtt{co}); [\mathtt{E}^{\mathtt{at}}]$*;*

2) *отсутствие буферизации при чтении (load buffering): отношение* $\mathtt{po} \cup \mathtt{rfe} \cup [\mathtt{E}^{\mathtt{at}}]; (\mathtt{coe} \cup \mathtt{fre}); [\mathtt{E}^{\mathtt{at}}]$ *ациклично.*

3.4.2. Промежуточная модель памяти

Промежуточная модель памяти (`IMM`) определена декларативно. Полный предикат согласованности `IMM` достаточно сложен, поэтому мы

рассмотрим лишь часть модели, которая будет необходима для построения схемы компиляции. Перед этим введём ещё несколько обозначений. R_{loc} будем обозначать рёбра R, проходящие между вершинами с метками одного и того же адреса, $R_{\neq loc}$ — между вершинами с метками разных адресов.

Синтаксис программ на IMM напоминает таковой в C++ — помимо инструкций атомарного чтения и записи есть инструкции барьеров памяти, а также операций read-modify-write. В граф исполнения программы на IMM, помимо отношений, перечисленных в разделе 3.3, также входит отношение, связывающее события чтения и записи, которые совершаются при операции read-modify-write:

$G.\mathtt{rmw} \subseteq ([G.\mathtt{R}]; \mathtt{po}|_{\text{imm}}; [G.\mathtt{W}])_{loc}$.

Для построения схемы компиляции мы пользуемся расширением [15] IMM, которое дополняет оригинальную модель [18] sc-операциями.

***Определение* 3.7.** *Сценарий исполнения называется* согласованным по IMM, *если в соответствующем графе исполнения выполняются следующие аксиомы:*

1) *отношение* $\mathtt{hb}; (\mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})^+$ *иррефлексивно, при этом справедливо следующее:*

   $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^+$

   $\mathtt{sw} \triangleq \mathtt{release}; (\mathtt{rfi} \cup \mathtt{po}^?_{loc}; \mathtt{rfe}); ([\mathtt{R}^{\mathtt{acq}}] \cup \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}])$

   $\mathtt{release} \triangleq ([\mathtt{W}^{\mathtt{rel}}] \cup [\mathtt{F}^{\mathtt{rel}}]; \mathtt{po}); \mathtt{rs}$

   $\mathtt{rs} \triangleq [\mathtt{W}]; \mathtt{po}_{loc}; [\mathtt{W}] \cup [\mathtt{W}]; (\mathtt{po}^?_{loc}; \mathtt{rfe}; \mathtt{rmw})^*$;

2) *операции read-modify-write являются атомарными:*

   $\mathtt{rmw} \cap (\mathtt{fre}; \mathtt{coe}) = \emptyset$;

3) *отношение* ar *ациклично, при этом справедливо следующее:*

   $\mathtt{ar} \supset \mathtt{rfe} \cup \mathtt{bob}$

   $\mathtt{bob} \supset [\mathtt{R}^{\mathtt{acq}}]; \mathtt{po} \cup \mathtt{po}; [\mathtt{F}] \cup [\mathtt{F}]; \mathtt{po}$;

4) отношение $\mathsf{psc}_{\text{base}}$ *ациклично, при этом справедливо следующее:*

$$\mathsf{psc}_{\text{base}} \triangleq ([\mathsf{E}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}^{?}); \mathsf{scb}; ([\mathsf{E}^{\mathsf{sc}}] \cup \mathsf{hb}^{?}; [\mathsf{F}^{\mathsf{sc}}])$$

$$\mathsf{scb} \triangleq \mathsf{po} \cup \mathsf{po}_{\not=loc}; \mathsf{hb}; \mathsf{po}_{\not=loc} \cup \mathsf{hb}_{loc} \cup \mathsf{co} \cup \mathsf{fr}.$$

4. Обоснование схемы компиляции

4.1. Необходимость введения дополнительных инструкций

Ранее в разделе 3.2 был рассмотрен пример программы, компиляция которой требует вставки дополнительных инструкций. Для удобства граф её исполнения приведён здесь на рис. 5. Именно, данный граф показывает, что ребро $\texttt{rfe}$ должно быть окружено барьерами `rel` и `acq`, поэтому инструкции неатомарной записи и атомарного чтения компилируются с использованием указанных барьеров.

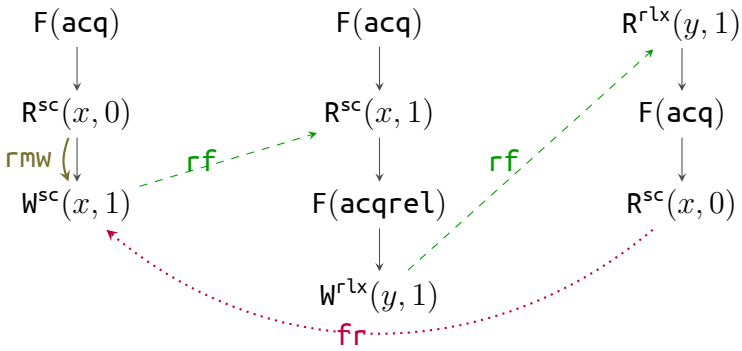

Рис. 5: Ребро $\texttt{rfe}$ между вершинами с режимом `rlx` должно быть окружено барьерами

Однако видно, что результирующая программа на рис. 4 получена с использованием схемы компиляции, которая вставляет в программу и другие инструкции. Так, барьер перед инструкцией неатомарной записи по адресу y имеет более строгий режим `acqrel`. Более того, инструкция атомарной записи компилируется в инструкцию read-modify-write, предварённую `acq`-барьером.

Приведённые ниже примеры демонстрируют, какие нежелательные сценарии поведения запрещают барьеры и инструкции read-modify-write. Для простоты мы рассмотрим эти случаи по отдельности, т.е. не будем вставлять барьеры в граф, содержащий read-modify-write и наоборот.

Использование барьеров позволяет обеспечить гарантируемое `OCaml MM` свойство multicopy atomicity для атомарных адресов. Это означает

отсутствие в графе исполнения скомпилированной программы циклов, состоящих из рёбер fr, проходящих между вершинами с меткой sc, а также рёбер po и rf. IMM в общем случае допускает такие циклы. Поскольку отношение ar в IMM не учитывает рёбра fr, то с целью запрета подобного сценария поведения цикл вида $[\mathtt{W}^{\mathtt{at}}]; (\mathtt{po}; \mathtt{rf})^{+}; \mathtt{po}; [\mathtt{R}^{\mathtt{at}}]; \mathtt{fr}$ необходимо представить как $\mathtt{hb}^{+}; \mathtt{fr}$. Для этого каждое ребро rf между неатомарными вершинами необходимо окружить барьерами. Кроме того, так как после подобного ребра может идти rf между атомарными вершинами, перед $\mathtt{W}^{\mathtt{at}}$ необходимо также расположить acq-барьер. Наконец, acq-барьер необходимо расположить и перед ребром fr, т.е. перед событием атомарного чтения. рис. 6 показывает последовательность рёбер rf между вершинами различных типов, иллюстрирующую необходимость расположения барьеров.

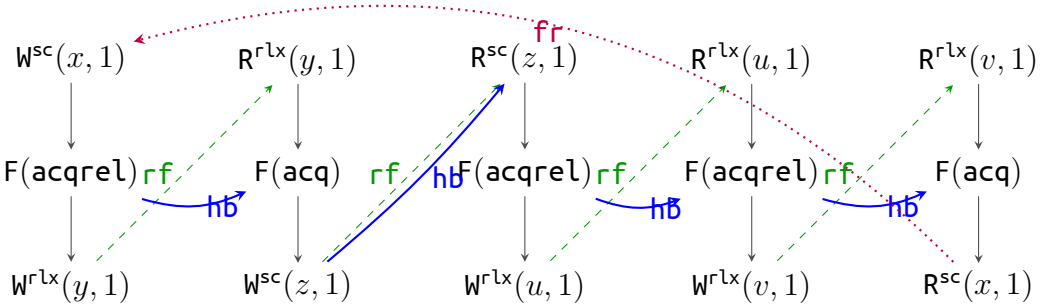

Рис. 6: Вставка барьеров позволяет провести рёбра hb вдоль рёбер rf и запретить указанный сценарий поведения

Компиляция инструкций атомарной записи в инструкции exchange позволяет, согласно требованиям OCamlMM, связать отношением hb события атомарной записи, упорядоченные по co. Вставка барьеров в этом случае не поможет, т.к. с помощью них можно провести hb лишь вдоль рёбер rf. Однако такое ребро можно ввести искусственно, добавив перед событием записи событие чтения, которое наблюдает co-предыдущую запись. Такое условие обеспечивается инструкцией exchange. Она порождает в графе пару вершин, связанных отношением rmw, что иллюстрируется на рис. 7.

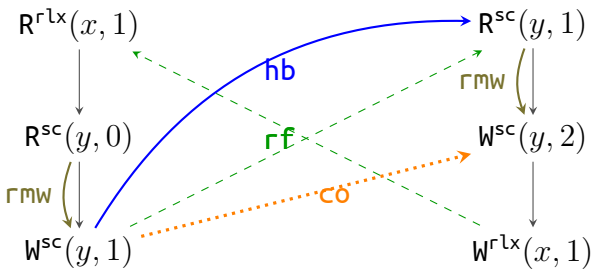

Рис. 7: Реализация атомарной записи инструкцией read-modify-write позволяет провести ребро hb и запретить указанный сценарий поведения

4.2. Схема компиляции

таб. 1 содержит предлагаемую схему компиляции OCamlMM в IMM. За основу взята схема компиляции OCamlMM в модель ARMv8 из [4].

Как описано в разделе 4.1, дополнительные инструкции используются для того, чтобы запретить для скомпилированной программы сценарии поведения, разрешённые IMM и запрещённые OCamlMM.

OCamlMM	IMM	ARMv8
$r := [x]^{\mathsf{na}};$	$r := [x]^{\mathsf{rlx}};$	$r := [x]^{\mathsf{rlx}};$
$[x]^{\mathsf{na}} := v;$	$\mathsf{fence}^{\mathsf{acqrel}}; [x]^{\mathsf{rlx}} := v;$	$\mathsf{fence}^{\mathsf{acq}}; [x]^{\mathsf{rlx}} := v;$
$r := [x]^{\mathsf{at}};$	$\mathsf{fence}^{\mathsf{acq}}; r := [x]^{\mathsf{sc}};$	$\mathsf{fence}^{\mathsf{acq}}; r := [x]^{\mathsf{sc}};$
$[x]^{\mathsf{at}} := v;$	$\mathsf{fence}^{\mathsf{acq}}; \mathsf{exchange}^{\mathsf{sc}}(\mathsf{x}, \mathsf{v});$	$\mathsf{exchange}^{\mathsf{sc}}(\mathsf{x}, \mathsf{v}); \mathsf{fence}^{\mathsf{rel}};$

Таблица 1: Схема компиляции OCamlMM в IMM и сравнение её со схемой компиляции OCamlMM в ARMv8

5. Доказательство корректности компиляции

***Теорема* 3.1.** *Пусть* $PO = ||_{\tau \in \mathsf{Tid}} PO_\tau$ *и* $PI = ||_{\tau \in \mathsf{Tid}} PI_\tau$ *— программы для моделей* OCamlMM *и* IMM *соответственно, причём для любого* $\tau \in \mathsf{Tid}$ *подпрограмма* PI_τ *получена компиляцией* PO_τ *с помощью схемы компиляции из таб. 1. Пусть* G_I *— согласованный по* IMM *граф исполнения* PI*. Тогда существует* G_O *— согласованный по* OCamlMM *граф исполнения* PO*, соответствующий* G_I*.*

Доказательство. В теореме 5.2 доказывается, что существует G_O, являющийся графом исполнения PO и соответствующий G_I (формальное понятие соответствия графов рассматривается в разделе 5.1).

Далее, в теореме 5.1 показывается, что согласованность по OCamlMM G_O следует из согласованности G_I по OCamlMM. Затем в теоремах 5.5 и 5.12 доказываются два условия согласованности G_I по OCamlMM. □

5.1. Соответствие графов исполнения

Необходимо точно определить соответствие графов исполнения, чтобы один и тот же сценарий исполнения программы можно было представить как в IMM, так и в OCamlMM. Отметим, что графы исполнения в этих моделях различны, т.к. в результате компиляции в программу добавляются новые инструкции, а в граф — новые вершины. Но эти различия не меняют сценарий поведения программы согласно определению 3.4.

Чтобы формализовать соответствие графов исполнения, опишем, чем отличаются множества вершин и рёбер в графах исполнения скомпилированной программы и исходной. Вершины графа исполнения в OCamlMM — те же, что в графе IMM, за исключением вершин-барьеров, а также операций чтения, выполняемых в ходе операции read-modify-write. При удалении этих вершин также удаляются смежные им рёбра. Метки оставшихся событий в OCamlMM совпадают с таковыми в IMM с точностью до переименования режимов доступа.

***Определение* 5.1.** *Граф исполнения по* OCamlMM G_O соответствует *графу исполнения по* IMM G_I*, если выполняются следующие условия:*

1) $G_O.\mathsf{E} = G_I.\mathsf{E} \cap (G_I.\mathsf{R} \cup G_I.\mathsf{W} \setminus dom(G_I.\mathtt{rmw}))$
2) $\forall e.\ e \in G_O.\mathsf{E} \Rightarrow G_O.\mathsf{Lab}\ e = \mathtt{renameMode}(G_I.\mathsf{Lab}\ e)$*, где* renameMode *меняет метку вершины с* sc *на* at *и с* rlx *на* na
3) $G_O.\mathtt{co} = [G_O.\mathsf{E}]; G_I.\mathtt{co}; [G_O.\mathsf{E}]$
4) $G_O.\mathtt{rf} = [G_O.\mathsf{E}]; G_I.\mathtt{rf}; [G_O.\mathsf{E}]$

***Теорема* 5.1.** *Пусть G_O и G_I — пара соответствующих графов. Тогда из согласованности G_I по* OCamlMM *следует согласованность G_O по* OCamlMM*.*

Доказательство. Условие согласованности по OCamlMM требует ациклич-ности и иррефлексивности отношений, построенных из рёбер po, rf, co и fr. Заметим, что $G_O.\mathsf{E} \subseteq G_I.\mathsf{E}$, так как в $G_O.E$ отсутствуют вершины, соответствующие барьерам и операциям read-modify-write, а новых вершин в G_O не вводится. Кроме того, при удалении вершин из графа удаляются и смежные с ними рёбра, поэтому выполнено следующее: $\forall r \in \{\mathsf{po}, \mathtt{rf}, \mathtt{co}, \mathtt{fr}\}.\ G_O.r \subseteq G_I.r$.

Если некоторое отношение ациклично (иррефлексивно), то и любое включённое в него отношение также ациклично (иррефлексивно). Поэтому из согласованности G_I по OCamlMM (при замене режимов доступа IMM на их аналоги в OCamlMM) следует согласованность G_O по OCamlMM. □

5.2. Построение графа, соответствующего данному

Пусть $PO = ||_{\tau \in \mathsf{Tid}} PO_\tau$ и $PI = ||_{\tau \in \mathsf{Tid}} PI_\tau$ — программы для моделей OCamlMM и IMM соответственно, причём для любого $\tau \in \mathsf{Tid}$ подпрограмма PI_τ получена компиляцией PO_τ с помощью схемы компиляции из таб. 1. Пусть G_I — согласованный по IMM граф исполнения PI.

Следующая теорема практически повторяет теорему 3.1, за исключением требования на согласованность по OCamlMM искомого графа.

***Теорема* 5.2.** *Существует граф G_O, который задаёт сценарий исполнения PO и соответствует G_I.*

Доказательство. Построим G_O по лемме 5.3.

Как следует из формулировки данной леммы, сужение этого графа на каждый из потоков является однопоточным графом исполнения соответствующей подпрограммы, поэтому G_O является графом исполнения PO. Осталось доказать соответствие G_O и G_I.

Условия на соответствие отношений `rf` и `co` выполняются непосредственно по формулировке леммы 5.3.

Заметим, что условия на соответствие множеств вершин $G_O.\mathsf{E}$ и $G_I.\mathsf{E}$ и функций $G_O.\mathsf{Lab}$ и $G_I.\mathsf{Lab}$ можно доказывать для сужений графов на потоки: эти условия выполняются для сужения на любой поток тогда и только тогда, когда они выполняются и во всём графе. По формулировке леммы 5.3 сужения этих графов на любой поток соответствуют друг другу. □

***Лемма* 5.3.** *Существует граф G_O, для которого справедливы следующие утверждения:*

- $G_O.\mathsf{Init} = G_I.\mathsf{Init}$;
- $G_O.\mathtt{co} = [G_O.\mathsf{E}]; G_I.\mathtt{co}; [G_O.\mathsf{E}]$;
- $G_O.\mathtt{rf} = [G_O.\mathsf{E}]; G_I.\mathtt{rf}; [G_O.\mathsf{E}]$;
- *для всех $\tau \in \mathsf{Tid}$ сужение $G_{O\tau}$ соответствует $G_{I\tau}$, а также является графом исполнения подпрограммы PO_τ.*

Доказательство. Искомый граф построим как объединение однопоточных графов исполнения, полученных по лемме 5.4. К этому объединению добавим множество инициализирующих вершин $G_I.\mathsf{Init}$. Отношение `co` и `rf` зададим как таковые в G_I, ограничив их на множества вершин в G_O. □

***Лемма* 5.4.** *Для всех $\tau \in Tid, PO_\tau, PI_\tau, G_{I\tau}$, где $G_{I\tau}$ является сужением G_I на поток τ, существует такой граф $G_{O\tau}$, который является графом исполнения для программы PO_τ и соответствует $G_{I\tau}$.*

Доказательство. Сужение G_{I_τ} является однопоточным графом исполнения для соответствующего потока, так как G_I — граф исполнения для программы PI. Как упоминалось в разделе 3.3, для задания этого используется операционная семантика, которая при исполнении инструкций потока строит соответствующий граф. Состояние абстрактной машины такой семантики содержит текущий граф, значения регистров и указатель на текущую инструкцию.

Для упрощения доказательства сгруппируем переходы абстрактной машины, соответствующие исполнению блоков инструкций, каждый из которых получен компиляцией одной инструкции в PO_τ. Мы получаем блочные состояния и переходим к т.н. блочному исполнению, в ходе которого выполнение блока переходов рассматривается как неделимый переход.

Исходное утверждение доказывается индукцией. Именно, для любого числа блочных переходов и результирующего состояния IMM-машины можно исполнить столько же обычных переходов исполнения PO_τ и получить состояние OCamlMM-машины, соответствующее заданному. □

5.3. Доказательство последовательной согласованности по отдельным адресам

***Теорема* 5.5.** *Отношение* $\mathsf{hbo}; (\mathsf{co} \cup \mathsf{fr})$ *является иррефлексивным.*

Докажем, что для выбранной схемой компиляции справедливо $\mathsf{hbo} \subseteq \mathsf{hb}$. С учётом этого доказательство теоремы тривиально: по определению согласованности по IMM, отношение $\mathsf{hb}; (\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})$ иррефлексивно.

Для этого факта сначала покажем, что последовательность рёбер co между sc-событиями порождает hb.

***Лемма* 5.6.** *Порядок событий* sc*-записи согласуется с отношением happens-before:* $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{co}; [\mathsf{E}^{\mathsf{sc}}] \subseteq \mathsf{hb}$.

Доказательство. Заметим, что $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{co}; [\mathsf{E}^{\mathsf{sc}}]$ транзитивно. Тогда можно перейти к рассмотрению непосредственных co-соседей.

Предположим, что $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{co}_{imm}; [\mathsf{E}^{\mathsf{sc}}] \subseteq [\mathsf{E}^{\mathsf{sc}}]; \mathsf{rf}; [\mathsf{E}^{\mathsf{sc}}]; \mathsf{po}$. Тогда доказательство тривиально: rf по sc событиям порождает hb, как и следующий за ним po. Значит, остаётся доказать утверждение о включении в $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{rf}; [\mathsf{E}^{\mathsf{sc}}]; \mathsf{po}$.

Рассмотрим два события $w_1, w_2 \in \mathsf{W}^{\mathsf{sc}}$ — непосредственных co-соседей. По схеме компиляции перед w_2 следуют $f \in \mathsf{F}^{\mathsf{acq}}$ и $r \in \mathsf{R}^{\mathsf{sc}}$, причём $\mathsf{rmw}(r, w_2)$.

Покажем, что $\mathsf{rf}(w_1, r)$. Рассмотрим событие записи w', из которого читает r. Обращения по одному и тому же адресу имеют один и тот же режим, поэтому $w' \in \mathsf{W}^{\mathsf{sc}}$. Пусть $w_1 \neq w'$. Тогда либо $\mathsf{co}(w', w_1)$, либо, наоборот, $\mathsf{co}(w_1, w')$. В первом случае нарушается атомарность rmw между w_2 и r. Во втором случае получается, что между co-соседями w_1 и w_2 расположен w', что невозможно. □

***Лемма* 5.7.** *Отношение happens-before в* IMM *содержит happens-before в* OCamlMM: $\mathsf{hbo} \subseteq \mathsf{hb}$.

Доказательство. $\mathsf{hbo} \triangleq \mathsf{po} \cup [\mathsf{E}^{\mathsf{sc}}]; (\mathsf{co} \cup \mathsf{rf}); [\mathsf{E}^{\mathsf{sc}}]$. По предыдущей лемме co, ограниченный на sc, входит в hb. rf по sc событиями порождает sw, и, следовательно, hb. □

5.4. Доказательство отсутствия буферизации при чтении

Сначала докажем утверждения, которые позволят нам находить барьеры в программном порядке между событиями.

***Лемма* 5.8.** *Перед событием записи располагается барьер:*
$[\mathsf{E} \setminus \mathsf{F}]; \mathsf{po}; [\mathsf{W}] \subseteq \mathsf{po}; ([\mathsf{F}^{\mathsf{acqrel}}]; \mathsf{po}; [\mathsf{E}^{\mathsf{rlx}}] \cup [\mathsf{F}^{\mathsf{acq}}]; \mathsf{po}; [\mathsf{E}^{\mathsf{sc}}]); [\mathsf{W}] \cup \mathsf{rmw}$.

Доказательство. Согласно схеме компиляции, инструкция барьера располагается либо непосредственно перед инструкцией неатомарной записи, либо перед инструкцией read-modify-write. Таким образом, барьера между событием и po-следующим событием записи может не быть, только если это событие чтения в rmw. □

***Лемма* 5.9.** *В программном порядке между событиями* $\mathtt{rlx}$ *и* $\mathtt{sc}$ *располагается барьер:* $[\mathsf{E}^{\mathtt{rlx}}]; \mathtt{po}; [\mathsf{E}^{\mathtt{sc}}] \subseteq \mathtt{po}; [\mathsf{F}^{\mathtt{acq}}]; \mathtt{po}$.

Доказательство. Согласно схеме компиляции, все инструкции в режиме $\mathtt{sc}$ предваряются $\mathtt{acq}$-барьерами. □

Кроме того, нам понадобятся следующие факты из алгебры [21].

***Лемма* 5.10.** *Цикл из рёбер двух типов можно представить в виде чередующихся участков рёбер каждого типа:* $(\mathtt{x} \cup \mathtt{y})^+ = \mathtt{y}^+ \cup \mathtt{y}^*; (\mathtt{x}; \mathtt{y}^*)^+$*, где* $\mathtt{x}$, $\mathtt{y}$ *— произвольные отношения.*

***Лемма* 5.11.** *Отношение* $\mathtt{x} \cup \mathtt{y}$ *является ацикличным, если ацикличны отношения* $\mathtt{x}$, $\mathtt{y}$ *и* $\mathtt{x}^+; \mathtt{y}^+$.

***Теорема* 5.12.** *Отношение* $\mathtt{po} \cup \mathtt{rfe} \cup [\mathsf{E}^{\mathtt{sc}}]; (\mathtt{coe} \cup \mathtt{fre}); [\mathsf{E}^{\mathtt{sc}}]$ *является ацикличным.*

Сгруппируем первые два отношения в объединении. Тогда по лемме 5.11 нужно показать ацикличность следующих отношений:

- $\mathtt{po} \cup \mathtt{rfe}$;
- $[\mathsf{E}^{\mathtt{sc}}]; (\mathtt{coe} \cup \mathtt{fre}); [\mathsf{E}^{\mathtt{sc}}]$;
- $(\mathtt{po} \cup \mathtt{rfe})^+; ([\mathsf{E}^{\mathtt{sc}}]; (\mathtt{coe} \cup \mathtt{fre}); [\mathsf{E}^{\mathtt{sc}}])^+$, и это эквивалентно ацикличности: $[\mathsf{E}^{\mathtt{sc}}]; (\mathtt{po} \cup \mathtt{rfe})^+; [\mathsf{E}^{\mathtt{sc}}]; ([\mathsf{E}^{\mathtt{sc}}]; (\mathtt{coe} \cup \mathtt{fre}); [\mathsf{E}^{\mathtt{sc}}])^+$.

Докажем второе и третье утверждение показав, что соответствующие отношения лежат в $([\mathsf{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathsf{E}^{\mathtt{sc}}])^+ \subseteq \mathtt{psc}^+_{\text{base}}$. Напомним, что $\mathtt{scb} \triangleq \mathtt{po} \cup \mathtt{po}_{\neq loc}; \mathtt{hb}; \mathtt{po}_{\neq loc} \cup \mathtt{hb}_{loc} \cup \mathtt{co} \cup \mathtt{fr}$ и $\mathtt{psc}_{\text{base}} \triangleq ([\mathsf{E}^{\mathtt{sc}}] \cup [\mathsf{F}^{\mathtt{sc}}]; \mathtt{hb}^?); \mathtt{scb}; ([\mathsf{E}^{\mathtt{sc}}] \cup \mathtt{hb}^?; [\mathsf{F}^{\mathtt{sc}}])$. В свою очередь, ацикличность $\mathtt{psc}_{\text{base}}$ следует из согласованности графа по $\mathtt{IMM}$.

Теперь видно, что второе утверждение верно по определению $\mathtt{scb}$. По этой же причине для доказательства третьего утверждения остаётся показать, что $[\mathsf{E}^{\mathtt{sc}}]; (\mathtt{po} \cup \mathtt{rfe})^+; [\mathsf{E}^{\mathtt{sc}}] \subseteq ([\mathsf{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathsf{E}^{\mathtt{sc}}])^*$.

***Теорема* 5.13.** *Отношение* $\mathtt{po} \cup \mathtt{rfe}$ *ациклично.*

Доказательство. Вновь воспользуемся леммой 5.11 и разложим условие ацикличности объединения на ацикличность отношений $\mathtt{po}$, $\mathtt{rfe}$ и $\mathtt{po}^{+}; \mathtt{rfe}^{+}$. Ацикличность первого следует из согласованности по IMM, двух и более рёбер второго подряд идти не может (т.к. их концы имеют разные типы), а ацикличность третьего эквивалентна ацикличности $\mathtt{po}; \mathtt{rfe}$.

Пусть такой цикл существует. Покажем, что это противоречит условию ацикличности $\mathtt{ar}$ (следует из согласованности по IMM). Напомним, что $\mathtt{ar} \supseteq \mathtt{rfe} \cup \mathtt{bob}$ и $\mathtt{bob} \supseteq [\mathtt{R}^{\mathtt{acq}}]; \mathtt{po} \cup \mathtt{po}; [\mathtt{F}] \cup [\mathtt{F}]; \mathtt{po}$.

По лемме 5.8 перед событием записи, которой начинается ребро $\mathtt{rfe}$, есть барьер $\mathtt{F}^{\sqsupseteq \mathtt{acq}}$, либо весь $\mathtt{po}$ является $\mathtt{rmw}$. В первом случае внутри $\mathtt{po}$ есть барьер, а такое отношение лежит в $\mathtt{bob} \subseteq \mathtt{ar}$. Во втором случае $\mathtt{rmw}$ начинается с $\mathtt{R}^{\mathtt{sc}}$, и такое ребро $\mathtt{po} \supseteq \mathtt{rmw}$ также содержится в $\mathtt{bob}$. Наконец, $\mathtt{rfe} \subseteq \mathtt{ar}$. □

Теперь для доказательства второго условия согласованности по OCaml MM осталось доказать утверждение $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{po} \cup \mathtt{rfe})^{+}; [\mathtt{E}^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^{*}$.

***Теорема* 5.14.** *Последовательность из рёбер* $\mathtt{po}$ *и* $\mathtt{rfe}$ *между вершинами* $\mathtt{sc}$ *состоит из рёбер* $\mathtt{scb}$ *между вершинами* $\mathtt{sc}$*:* $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{po} \cup \mathtt{rfe})^{+}; [\mathtt{E}^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^{*}$.

Доказательство. Сначала сформулируем утверждение, которое позволит отбрасывать $\mathtt{rfe}$-рёбра.

***Лемма* 5.15.** *Рёбра* $\mathtt{rfe}$*, у которых один из концов —* $\mathtt{sc}$*, входят в* $\mathtt{scb}$*:* $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe} \cup [\mathtt{W} \setminus \mathsf{Init}]; \mathtt{rfe}; [\mathtt{E}^{\mathtt{sc}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}]$.

Доказательство. Если один из концов ребра $\mathtt{rf}$ является $\mathtt{sc}$, таким же является и второй (исключение — чтение из инициализирующих записей, которые в IMM являются $\mathtt{rlx}$). Тогда $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{rf}; [\mathtt{E}^{\mathtt{sc}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{hb}_{loc}; [\mathtt{E}^{\mathtt{sc}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}]$. □

По лемме 5.10 имеем следующее:

$[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{po} \cup \mathtt{rfe})^{+}; [\mathtt{E}^{\mathtt{sc}}] = [\mathtt{E}^{\mathtt{sc}}]; (\mathtt{rfe}^{+} \cup \mathtt{rfe}^{*}; (\mathtt{po}; \mathtt{rfe}^{*})^{+}); [\mathtt{E}^{\mathtt{sc}}] = [\mathtt{E}^{\mathtt{sc}}]; (\mathtt{rfe} \cup \mathtt{rfe}^{?}; (\mathtt{po}; \mathtt{rfe}^{?})^{+}); [\mathtt{E}^{\mathtt{sc}}]$.

Начальные участки $\mathtt{rfe}$, если они есть, можно отбросить по лемме 5.15. Рассмотрим оставшееся транзитивное замыкание:

$(\mathtt{po}; \mathtt{rfe}^?)^+ = \mathtt{po}; (\mathtt{po}; \mathtt{rfe})^*; \mathtt{po}^? = \mathtt{po}; \mathtt{po}^? \cup \mathtt{po}; (\mathtt{po}; \mathtt{rfe})^+; \mathtt{po}^? = \mathtt{po} \cup (\mathtt{po}; \mathtt{rfe})^+; \mathtt{po}^?$.

В первом случае отношение сводится к $\mathtt{po} \subseteq \mathtt{scb}$. Во втором случае, если последним ребром является $\mathtt{rfe}$, то его можно отбросить по лемме 5.15. Остаётся случай $(\mathtt{po}; \mathtt{rfe})^+; \mathtt{po}$. К каждой паре $\mathtt{po}; \mathtt{rfe}$ можно применить лемму 5.8. В результате нужно доказать следующее утверждение:

$[(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]; \quad (\mathtt{po}; ([\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}]; \mathtt{rfe} \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe}) \cup \mathtt{rmw}; \mathtt{rfe})^+ ; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; \mathtt{E}^{\mathtt{sc}}])^*$

Воспользуемся леммой 5.10. Имеем, что либо транзитивное замыкание состоит только из $\mathtt{rmw}; \mathtt{rfe}$, либо такие пары рёбер могут следовать после $\mathtt{po}; ([\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; (\mathtt{E}^{\mathtt{rlx}}); \mathtt{rfe} \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe})$. В первом случае замыкание имеет вид $\mathtt{hb}_{loc} \subseteq \mathtt{scb}$, а так как оно заканчивается $\mathtt{sc}$-событием, оставшееся ребро $\mathtt{po}$ также пройдёт по $\mathtt{sc}$ и образует $\mathtt{scb}$. Во втором случае рассмотрим, что именно находится под транзитивным замыканием:

$(\mathtt{po}; ([\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}]; \mathtt{rfe} \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe}); (\mathtt{rmw}; \mathtt{rfe})^*)^+ =$

$(\mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; ([\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}] \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]); (\mathtt{rfe}; \mathtt{rmw})^*; \mathtt{rfe})^+ =$

$(\mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; \mathtt{C}; \mathtt{rfe})^+$, где

$\mathtt{C} = \mathtt{C}_1 \cup \mathtt{C}_2 = [\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}]; (\mathtt{rfe}; \mathtt{rmw})^* \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]; (\mathtt{rfe}; \mathtt{rmw})^*$.

Заметим, что верно следующее:

$(\mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; \mathtt{C}; \mathtt{rfe})^+ = \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; (\mathtt{C}; \mathtt{rfe}; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}])^*; \mathtt{C}; \mathtt{rfe}$.

В результате необходимо доказать вот что:

$[(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; (\mathtt{C}; \mathtt{rfe}; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}];)^*; \mathtt{C}; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*$.

Заметим, что выполнено следующее утверждение:

$(\mathtt{C}; \mathtt{rfe}; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}])^* = \left(([\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}] \cup [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]); (\mathtt{rfe}; \mathtt{rmw})^*; \mathtt{rfe}; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]\right)^* \subseteq \mathtt{hb}^?$.

Это справедливо, поскольку $\mathtt{hb}$ задано так:

$\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^+$,

$\mathtt{sw} \supset \mathtt{release}; \mathtt{rfe}; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]$,

$\mathtt{release} \triangleq ([\mathtt{W}^{\mathtt{rel}}] \cup [\mathtt{F}^{\mathtt{rel}}]; \mathtt{po}); \mathtt{rs}$,

$\mathtt{rs} \supset (\mathtt{rfe}; \mathtt{rmw})^*$.

Вспомним, что $\mathtt{po}_{\neq loc}; \mathtt{hb}; \mathtt{po}_{\neq loc} \subseteq \mathtt{scb}$. Воспользуемся тем, что $\mathtt{po}$ между событием чтения/записи и барьером образует именно $\mathtt{po}_{\neq loc}$. Тогда видно, что $[(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}$.

Остаётся доказать следующее утверждение:

$[\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{C}; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*$.

Для этого перепишем $\mathtt{C} = \mathtt{C}_1 \cup \mathtt{C}_2$ и докажем утверждение для $\mathtt{C}_1$ и $\mathtt{C}_2$ по отдельности.

Случай с $\mathtt{C}_1$. Покажем, что

$[\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{C}_1; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*$. Заметим, что по лемме 5.9 в последнем ребре $\mathtt{po}$ найдётся $\mathtt{acq}$-барьер, с помощью которого можно будет построить ребро $\mathtt{hb}$:

$\mathtt{C}_1; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] = [\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}]; (\mathtt{rfe}; \mathtt{rmw})^*; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]$
$= [\mathtt{F}^{\mathtt{acqrel}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{rlx}}]; (\mathtt{rfe}; \mathtt{rmw})^*; \mathtt{rfe}; [\mathtt{E}^{\mathtt{rlx}}]; \mathtt{po}; [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]$
$\subseteq \mathtt{hb}; \mathtt{po}_{\neq loc}; [\mathtt{E}^{\mathtt{sc}}]$. Тогда $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{C}_1; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]$
$\subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{hb}; \mathtt{po}_{\neq loc}; [\mathtt{E}^{\mathtt{sc}}]$
$\subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}]$.

Случай с $\mathtt{C}_2$. Покажем, что выполнено следующее:

$[\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{C}_2; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}] \subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*$. Заметим, что последовательность пар рёбер $\mathtt{rfe}; \mathtt{rmw}$ входит в $\mathtt{scb}$:

$\mathtt{C}_2 = [\mathtt{F}^{\mathtt{acq}}]; \mathtt{po}; [\mathtt{E}^{\mathtt{sc}}]; (\mathtt{rfe}; \mathtt{rmw})^* \subseteq \mathtt{po}_{\neq loc}; [\mathtt{E}^{\mathtt{sc}}]; ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe}; [\mathtt{E}^{\mathtt{sc}}]; \mathtt{rmw}; [\mathtt{E}^{\mathtt{sc}}])^*; [\mathtt{E}^{\mathtt{sc}}]$
$\subseteq \mathtt{po}_{\neq loc}; [\mathtt{E}^{\mathtt{sc}}]; ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*; [\mathtt{E}^{\mathtt{sc}}]$. В этом случае имеем:
$[\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{C}_2; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]$
$\subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{po}_{\neq loc}; \mathtt{hb}^?; \mathtt{po}_{\neq loc}; [\mathtt{E}^{\mathtt{sc}}]; ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*; [\mathtt{E}^{\mathtt{sc}}]; \mathtt{rfe}; \mathtt{po}; [(\mathtt{W} \cup \mathtt{R})^{\mathtt{sc}}]$
$\subseteq ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}]); ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^*; ([\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}])^2$. □

6. Формализация доказательства в Coq

В некоторых опубликованных доказательствах корректности компиляции впоследствии были найдены неточности. Например, [10] демонстрирует ошибку в схеме компиляции C++ в Power, [18] — в схеме компиляции модели Promising в Power.

Чтобы избежать подобных ошибок, доказательство из раздела 5 было решено формализовать в Coq. Это одна из систем интерактивного доказательства теорем, которые позволяют записать доказательство математического утверждения на функциональном языке программирования и автоматически проверить его корректность. В настоящее время Coq и его аналоги используются как для формализации математики [6], так и для верификации отдельных программных проектов [12].

Выбор системы Coq обусловлен наличием существующей формализации `IMM` и доказательств корректности компиляции для неё [17]. Данная формализация построена на основе библиотеки `hahn`, упрощающей работу с бинарными отношениями.

На данный момент формализация `OCamlMM` и доказательства корректности компиляции в `IMM` занимает порядка 8 тыс. строк кода на Coq. Исходный код доступен на [17].

Структура доказательства в Coq повторяет таковую в разделе 5. Формализация разбита на следующие файлы:

- `OCaml.v` — формализация `OCamlMM`;
- `OmmProgram.v` — условия на программы в `OCamlMM` и исполнения в этой модели;
- `OmmImmCompScheme.v` — описание схемы компиляции и её свойств;
- `OmmImmCompCorrect.v` — высокоуровневая структура доказательства;
- `ImmImpliesOmm.v` — доказательство согласованности по `OCamlMM` графа исполнения, согласованного по `IMM`;

- `OmmImmSimulation.v` — доказательство отношения симуляции между исполнениями в `OCamlMM` и `IMM`;
- `GraphConstruction.v` — построение графа исполнения из однопоточных графов исполнения;
- `BlockSteps.v` — преобразование скомпилированной программы в последовательность блоков, полученных при компиляции одной исходной инструкции;
- `CompSchemeGraph.v` — доказательство утверждения о том, что в графе исполнения скомпилированной программы выполняются предпосылки, необходимые для доказательства его согласованности по `OCamlMM`;
- `BoundedRelsProperties.v` — вспомогательные утверждения о компонентах графа, значения которых устанавливаются пошагово в ходе его построения;
- `ClosuresProperties.v` — вспомогательные утверждения про транзитивное и транзитивно-рефлексивное замыкания;
- `Utils.v` — вспомогательные утверждения про отношения;
- `ListHelpers.v` — вспомогательные утверждения про списки, в том числе те, которые доказаны в Coq 8.10, но не добавлены в стандартную библиотеку Coq 8.09, используемую в проекте.

В основе Coq лежит интуиционистская логика, т.е. по умолчанию в нём не используется закон исключённого третьего. Это исключает неконструктивные доказательства, что, в частности, делает возможным экстракцию доказательств на Coq в программы на других языках. Однако, так как экстракция данного доказательства не требуется, а применение закона исключённого третьего значительно упрощает доказательство (и в некоторых случаях является необходимым), то было решено использовать эту аксиому.

Одним из достоинств Coq является возможность написания пользовательских тактик — инструкций, преобразующих текущую цель доказательства. Такие тактики могут быть частично автоматизированы, т.к. включать перебор различных вариантов доказательства прозрачно от пользователя. В рамках данного проекта было разработано несколько тактик, которые значительно сократили объём доказательств. Подходящей ситуацией для их применения является перебор случаев, выкладки в которых являются схожими, но отличаются в деталях. В этом случае разумным решением является разработка специализированной тактики, которая нивелирует различия разных случаев путём автоматизации.

В рамках данного проекта применение Coq позволило уточнить входные условия на программы, которые до начала работы над проектом явным образом не оговаривались. Так, явным образом выражено условие на разделение атомарных и неатомарных адресов на уровне программы для `OCamlMM`. Также от этой программы требуется выделить под инструкции `exchange` специальный регистр, который не будет используется в остальной части программы. Наконец, для упрощения части доказательства, касающейся блочных состояний и переходов было решено ограничить целевые адреса в инструкциях `ifgoto`, запретив переходы за пределы программы. Данное ограничение выглядит разумным, т.к. соответствующее преобразование может быть выполнено реальным компилятором.

7. Существующие исследования

Проблема корректности компиляции из OCamlMM и в IMM рассматривается и в других работах. Так, авторы OCamlMM [4] разработали для неё схемы компиляции в архитектуры x86 и ARMv8 [19]. Модель памяти x86 практически не отличается от SC [9], поэтому для корректной компиляции OCamlMM в x86 достаточно лишь реализовать инструкцию атомарной записи с использованием xchg. В схеме компиляции OCaml MM в ARMv8, в отличие от предложенной нами схемы, при компиляции инструкции неатомарной записи используется барьер F^{acq}, а не F^{acqrel}. Это объясняется тем, что в модели ARMv8 отношение ob (аналог ar в IMM) включает в себя $rfe \cup fre \cup coe$ по неатомарным операциям и po с acq-барьером перед событием записи, поэтому в последовательности рёбер вида $(po; rfe)^+$ не требуется rel-барьер. Наконец, в [5] утверждается, что свойство локальной свободы от гонок можно реализовать в программной транзакционной памяти с использованием тех же схем компиляции в x86 и ARMv8.

В [18] приведены схемы компиляции моделей Promising [8] и RC11 [10] в IMM. Предикат консистентности RC11 требует ацикличности отношения $po \cup rf$, поэтому между каждым событием не-sc чтения и последующим событием записи необходимо располагать барьер. С этим дополнительным условием доказательство корректности компиляции значительно упрощается. В свою очередь, модель Promising является достаточно слабой, поэтому для неё корректной является тривиальная схема компиляции (при выполнении некоторых условий на инструкции read-modify-write в ней). Однако доказательство этого значительно сложнее, так как в нём применяется метод обхода графа исполнения, который позволяет связать операционную семантику модели Promising и декларативную семантику IMM.

Стоит отметить, что компиляция с использованием промежуточного представления программы может быть менее эффективной, чем компиляция в целевую модель памяти напрямую. Так, при композиции схем компиляции OCamlMM в IMM 1 и IMM в ARMv8 [15] инструкция

атомарной записи `OCamlMM` компилируется с использованием двух барьеров, в то время как схема компиляции `OCamlMM` в ARMv8 из [4] использует только один.

8. Заключение

В данной работе представлена схема компиляции OCamlMM в промежуточную модель, позволяющая получить схему компиляции OCamlMM в Power. При этом были получены следующие результаты.

- Предложена схема компиляции OCamlMM в IMM, использующая барьеры памяти и инструкции read-modify-write для необходимой синхронизации доступов к памяти.
- Была доказана корректность предложенной схемы. Для этого было введено формальное понятие соответствия графов, которое позволило связать сценарии исполнения исходной и скомпилированной программ. Затем было доказано, что для каждого согласованного по IMM графа исполнения скомпилированной программы существует соответствующий ему граф исполнения исходной программы, согласованный по OCamlMM.
- Полученное доказательство было формализовано в Coq. Без учёта существующей формализации IMM формализация заняла порядка 8 тыс. строк кода.

Результаты работы были представлены на Открытой конференции ИСП РАН и опубликованы в ”Трудах Института системного программирования РАН” [23].

Данное исследование может быть продолжено при реализации свойства локальной свободы от гонок в других моделях памяти. Например, изучается возможность реализовать это свойство в модели памяти C++ [3]. Результаты, полученные в данной работе, могут быть использованы для построения схем компиляции таких моделей. В частности, сравнение схемы компиляции OCamlMM в ARMv8 [4] и полученной в данной работе позволяет предположить, что с большой вероятностью для компиляции инструкции атомарной записи в подобных моделях необходимо будет использовать инструкцию атомарной замены.

Список литературы

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014), 7:1–7:74. ISSN: 0164-0925. DOI: 10.1145/2627752. URL: http://doi.acm.org/10.1145/2627752.

[2] Mark Batty et al. “Mathematizing C++ Concurrency”. In: *POPL 2011*. ACM, 2011, pp. 55–66. DOI: 10.1145/1925844.1926394.

[3] Simon Doherty. *Local Data-race Freedom and the C11 Memory Model*. Surrey Concurrency Workshop. 2019.

[4] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. “Bounding Data Races in Space and Time”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018.

[5] Brijesh Dongol, Radha Jagadeesan, and James Riely. “Modular Transactions: Bounding Mixed Races in Space and Time”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP ’19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 82–93. ISBN: 9781450362252. DOI: 10.1145/3293883.3295708. URL: https://doi.org/10.1145/3293883.3295708.

[6] Georges Gonthier. “The Four Colour Theorem: Engineering of a Formal Proof”. In: *Computer Mathematics: 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 333. ISBN: 9783540878261. URL: https://doi.org/10.1007/978-3-540-87827-8_28.

[7] IBM Power Systems. *IBM Power Systems Facts and Features:Enterprise and Scale-out Systems with POWER8 Processor Technology*. https://www.ibm.com/downloads/cas/JDRZDG0A. 2018.

[8] Jeehoon Kang et al. “A Promising Semantics for Relaxed-memory Concurrency”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 175–189. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009850. URL: http://doi.acm.org/10.1145/3009837.3009850.

[9] Ori Lahav and Viktor Vafeiadis. “Explaining Relaxed Memory Models with Program Transformations”. In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald et al. Cham: Springer International Publishing, 2016, pp. 479–495. ISBN: 978-3-319-48989-6.

[10] Ori Lahav et al. “Repairing Sequential Consistency in C/C++11”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 618–632. ISSN: 0362-1340. DOI: 10.1145/3140587.3062352. URL: http://doi.acm.org/10.1145/3140587.3062352.

[11] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers C-28* 9 (1979), pp. 690–691. URL: https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/.

[12] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[13] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java Memory Model”. In: *POPL 2005*. ACM, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336.

[14] Batty Mark et al. *C/C++11 mappings to processors*. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. 2016.

[15] Evgenii Moiseenko et al. “Reconciling Event Structures with Modern Multiprocessors”. In: (Nov. 2019), Accepted for European Conference on Object–Oriented Programming 2020.

[16] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407. ISBN: 978-3-642-03359-9.

[17] Anton Podkopaev, Ori Lahav, and Egor Namakonov. *Intermediate Memory Model*. https://github.com/weakmemory/imm. 2019.

[18] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. “Bridging the Gap Between Programming Languages and Hardware Weak Memory Models”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 69:1–69:31. ISSN: 2475-1421. DOI: 10.1145/3290382. URL: http://doi.acm.org/10.1145/3290382.

[19] Christopher Pulte et al. “Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 19:1–19:29. ISSN: 2475-1421. DOI: 10.1145/3158107. URL: http://doi.acm.org/10.1145/3158107.

[20] Coq development team. *The Coq Proof Assistant*. https://coq.inria.fr/.

[21] Viktor Vafeiadis, Ori Lahav, and Anton Podkopaev. *Hahn library*. https://github.com/vafeiadis/hahn/. 2020.

[22] Conrad Watt et al. “Repairing and mechanising the JavaScript relaxed memory model”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020). DOI: 10.1145/3385412.3385973. URL: http://dx.doi.org/10.1145/3385412.3385973.

[23] Егор Намаконов и Антон Подкопаев. «Компиляция модели памяти OCaml в Power». В: *Труды ИСП РАН* 31.5 (2019), с. 63—78. DOI: 10.15514/ISPRAS-2019-31(5)-4.