博士学位论文

中国科学技术大学
博士毕业论文
基于操作语义的弱内存模型描述及程序逻辑研究

作者姓名：张扬
学科专业：计算机软件与理论
导师姓名：冯新宇 教授

完成时间：二〇一五年五月

University of Science and Technology of China

A dissertation for doctor's degree

An Operational Relaxed Memory Model and Program Logic for Concurrency Verification

Author : Yang Zhang

Speciality : Computer Software and Theory

Supervisor : Prof. Xinyu Feng

Finished Time : May, 2015

基于操作语义的弱内存模型描述及程序逻辑研究

十一系　张扬　中国科学技术大学

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文, 是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☐ 公开 ☐ 保密 _____ 年

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘 要

程序语言的内存模型规定了在程序执行的过程中内存访问是如何发生的。它作为桥梁将为程序员和语言实现连接起来，帮助程序员写出正确的并发程序。在现实世界中，大多数的硬件和编译系统都是基于弱内存模型的假设，即内存访问并不是严格按照程序顺序执行，以用来支持各类优化。本文研究了弱内存模型的设计，并提出了可以支持在弱内存模型上进行程序验证的程序逻辑。具体来说，本文在弱内存模型和程序逻辑方面做出了如下的贡献:

首先，本文提出一种新的弱内存模型 OHMM, 这是 Happens-before 内存模型 (HMM) 的变种。这个模型通过对一个简单语言赋予具体的操作语义，并通过它在抽象机上的程序行为来模拟 HMM。由于 OHMM 所允许的程序行为是通过操作语义自然生成的，所以它自然而然的避免了所谓的凭空出现 (out-of-thin-air) 的程序行为。另外一方面，OHMM 使用一种我们称之为重放的机制来允许某些符合一定条件的指令在抽象机上能够多次执行，来模拟现实世界中编译器和处理器优化中的投机执行和优化。总的来说，我们的模型对于无锁程序的约束会比 Java 内存模型 (JMM) 更加弱一些，因此我们将会允许更多的编译器优化算法在我们的模型上能够使用。同时，在 OHMM 上，程序行为在直观上会比 JMM 更加自然。许多在 JMM 上可能出现但是明显违反直观认识的程序，在我们的模型上就不再合法。我们希望 OHMM 可以成为可供类 Java 语言选择的一种新内存模型。

其次, 本文提出一种新的用于验证并发程序在 TSO(Total Store Order) 弱内存模型下正确性的程序逻辑。TSO 模型所允许的弱行为是 OHMM 的子集。我们知道，TSO 模型已经被用作 X86 和 SPARC-TSO 处理器族的模型基础，并且在一些高级语言中也正在被提案作为其内存模型的基础。我们的逻辑对 LRG(Local Rely-Guarantee) 进行扩展，对其加入了关于 TSO 写缓存的断言, 这可以让我们对 TSO 模型中对外部线程不可见的局部的写缓存的状态进行描述。如同 LRG 一样，我们的程序逻辑支持对细粒度并发具有表达力强的 rely/guarantee 推理以及分离逻辑中的局部推理。同时，我们在逻辑上对 TSO 模型进行进一步抽象，把 TSO 共享内存分为 local 和 shared 两部分，这可以允许我们可以将那些在访问时只有单个线程能够访问的内存单元（逻辑上等同于 local 单元）的写操作直接写入内存，不需要经过写缓存。我们使用这个逻辑证明了一些具有代表性的并发算法在 TSO 上的正确性，包括 Peterson’s lock 算法, Simpson’s four slot 算法, concurrent GCD 算法以及 lock 的优化实现算法。

关键词： 弱内存模型, 程序验证, 并发, 程序逻辑

ABSTRACT

A memory model of a programming language specifies how memory accesses are made during program execution. It serves as a contract between programmers and the language implementation. In real world, most of architectures and compilers are designed based on the assumption of relaxed memory models, which means memory accesses doesn't rigorously follow the program order. Relaxed memory models can support many optimizations. This dissertation explores a new relaxed memory model and proposes a new program logic for verifying concurrent algorithms in relaxed memory model.

First, it presents OHMM, an operational variation of Happens-before Memory model(HMM). The model is specified by giving an operational semantics to a language running on an abstract machine designed to simulate HMM. Thanks to its generative nature, the model naturally prevents out-of-thin-air reads. On the other hand, it uses a novel replay mechanism to allow instructions to be executed multiple times, which can be used to model many useful speculations and optimization. The model is weaker than JMM for lockless programs, thus can accommodate more optimization, such as the reordering of independent memory accesses that is not valid in JMM. Program behaviors are more natural in this model than in JMM, and many of the anti-intuitive examples in JMM are no longer valid here. We hope OHMM can serve as the basis for new memory models for Java-like languages.

Second, it proposes a new program logic for verifying concurrent algorithms in the TSO (Total Store Order) memory model, which has been used as the basis for x86 and SPARC-TSO processor families, and in proposals for high-level programming languages. The behaviors of programs running on TSO model are a subset of one on OHMM。Our logic extends LRG with explicit assertions for write buffers in TSO, which can specify locally buffered writes that are invisible to other threads yet. Like LRG, the logic supports both expressive rely/guarantee style reasoning for fine-grained concurrency and small-footprint local reasoning as in separation logic. The distinction of local memory from shared also gives us a more abstract and simplified view of TSO, in which local writes are done directly to memory and not buffered. We have applied the logic to verify several representative algorithms in TSO, including Peterson's lock algorithm, Simpson's four slot algorithm, a concurrent GCD algorithm and an optimized implementation of locks in TSO.

Keywords: Relaxed memory model, Verification, Concurrency, Program Logic

目 录

插图索引

主要符号对照表

HMM	Happens-before 内存模型 (Happens-before Memory Model)
JMM	Java 内存模型 (Java Memory Model)
DRF	无数据竞争 (Data Race Freedom)
TSO	TSO 内存模型 (Total Store Order Memory Model)
OHMM	OHMM 内存模型 (Operational Happens-before Memory Model)
ATSO	TSO 内存模型变种
LRG	Local Rely-Guarantee Reasoning

第一章　绪论

随着多核处理器的广泛应用，并发程序设计已经由一种特殊的、只需少数高端技术人才掌握的技巧变为一种大多数程序员都应掌握的基本技能。为了确保写出正确的并发程序，程序员必须准确理解并发程序的运行模式（又称为程序语义）。而内存一致性模型 (memory consistency model) 描述了并发程序访问内存的方式，是理解并发程序语义的关键。然而，大多数习惯了串行程序设计的程序员仅仅了解最理想化的内存一致性模型，该模型与现实中各种模型有着很大差异。虽然这种差异对串行程序没有影响，但在并发环境下将导致程序行为的不同，直接影响程序的正确性 [1, 2]。因此内存一致性模型的准确描述和理解对于并发语言的设计和实现，以及并发软件开发和验证均具有重要意义。

处理器通过对内存的读写进行数据的访问和操作，而多处理器之间的通信则通过内存共享来完成。我们把程序运行过程中内存读写的执行方式称为**内存一致性模型**。它可以视为程序员和语言实现的接口，来帮助程序员更好的写出正确的并发程序。最被大家知晓的内存模型是顺序一致性模型 (Sequential Consistency model)，由 Lamport [3] 在 1979 提出。这个模型要求在同一个时间点只有一个内存操作发生，并且每个线程运行的时候都是严格按照程序顺序执行（program-order）的方式进行内存访问（具体参照第二章中的节 2.1）。

然而在现实世界中，实现顺序一致性模型的代价太过于昂贵，为了保证顺序一致的要求，便不得不禁止很多在硬件和编译器中有用的优化。这些优化在设计的时候，都能够保证在优化前后串行程序的行为不变，但是在并发环境下面，就可能会产生非预期的行为 [2]。举一个简单的例子，在下面的程序给出一个经典的实现并发线程互斥的算法 [4]。我们用 T1||T2 表示线程 T1 和 T2 可以分别被不同的处理核同时运行。x 和 y 是位于共享内存上的非同步变量，对它们的访问通过内存存取来完成，而 r 则代表的线程局部变量（寄存器）。该算法的目标是保证左右两个线程能够互斥访问临界区，即两个线程不会同时进入临界区。如果我们假设处理器运行单个线程的时候是严格按照程序给出的语句顺序逐条执行（即顺序执行），那么不难看出上面的算法确实实现了互斥功能：当线程 T1 和 T2 执行到 if 语句的时候，r_1 和 r_2 不可能同时为 0，所以最多只有一个线程能进入临界区。但上述分析是基于一种理想化的模型。现实中，编译器和处理器内部进行的优化都会导致内存操作的实际执行顺序和程序中的语句顺序的不一致（即乱序执行）。在上面的例子中，由于每个线程的前两条语句之间不具有数据依赖性，很多编译器和处理器都可能会反转它们的执行顺序，从而导致两个线程的 if 语句判断同时为真，使得两个线程同时进入临界区 [1, 2], 从而破坏了算法互斥访问的保证。

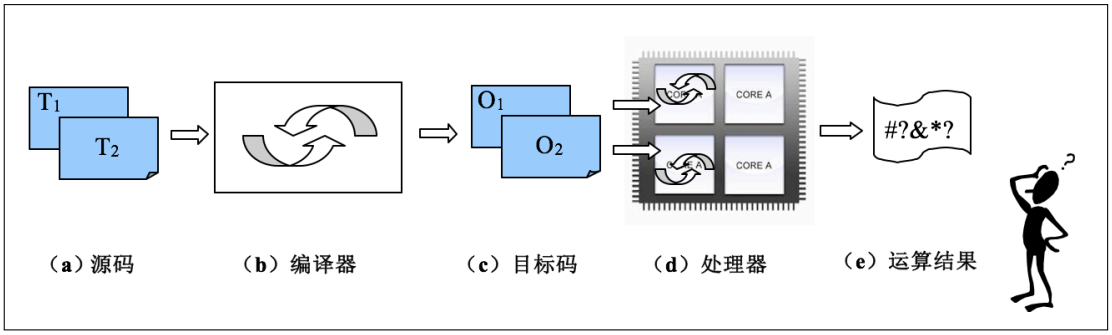


图 1.1: 并发程序的编译和执行过程（我们用环形剪头表示对内存操作进行乱序执行）

例 1.0.1. *Initially* $x = y = 0$.

1：	$x := 1;$	6：	$y := 1;$
2：	$r_1 := y;$	7：	$r_2 := x;$
3：	**if**$(r_1 == 0)\{$	8：	**if**$(r_2 == 0)\{$
4：	*critical region*	9：	*critical region*
5：	$\}$	10：	$\}$

Result: $r_1 = r_2 = 0$*?*

允许这种经过优化而产生的不符合顺序一致性的行为的模型我们称之为弱内存模型。虽然大多数程序员假定并发程序的运行满足顺序一致性，但现实中几乎所有的并发程序都在某种弱内存模型下运行，而且不同并发语言和处理器的内存模型各不相同。因此内存一致性模型作为系统实现和程序员之间的接口，对于处理器体系结构的实现、并发语言编译器的实现以及并发程序的开发和验证有着**重要的意义**:

- 从处理器体系结构的角度看，流水线和缓冲区的使用会导致线程的乱序执行（图 1.1(d) 所示）。处理器厂商需要准确描述可能的指令执行顺序（即处理器的内存模型），使得用户能够正确预期图 1.1中 (c)(e) 的关系，从而正确地生成机器指令。

- 从并发语言的编译器的实现角度看，一方面编译器的优化算法会调整源程序中的内存操作顺序（图 1.1(b)），使得 (a) 和 (c) 中的顺序不一致；另一方面，产生的机器码的执行顺序又可能被处理器来进一步改变（图 2(d)）。并发语言的设计和实现必须考虑到这两种情况及其效果的叠加，对源程序的可能表现出的行为进行准确描述（即并发语言的内存模型），使得程序员能够正确预期中图 1.1(a) 和 (e) 的关系，从而进行正确编程。

- 从程序员的角度看，理想化的顺序一致性模型并不足以解释图 1.1中源程序 (a) 和运算结果 (e) 之间的关系。程序员在实现如图 1 所示的并发算法的时候，要根据程序设计语言对内存模型的描述，考虑到弱内存模型下乱序执行对程序正确性的影响。

同时，弱内存模型的引入，也会导致并发程序的验证更加困难。因为弱内存模型使得并发程序的行为变得更加复杂，所以程序也更加容易出错。由于缺乏对弱内存模型的有效形式描述，以往的并发程序的验证逻辑，包括并发分离逻辑（CSL） [5], Rely-Guarantee 方法 [6] 及其各种变体 [7–9]，都是基于理想化的顺序一致性模型，因而均无法发现图 1 中的错误。基于模型检查（model checking）的验证方法本身受限于有限状态空间的要求，对能够验证的程序和性质有很多限制 [10]。

然而，现有的对内存模型的研究，往往存在这么几条缺陷：（1）仅提供模型的非形式化描述，往往具有歧义且不完整；（2）描述仅仅从处理器和编译器的实现者的角度给出，没有考虑程序员的角度，因而对并发程序的开发及验证帮助不大；(3) 对于常见的串行编译优化算法，内存模型往往不能保持其正确性;（4）对并发程序在弱内存模型下的正确性尚缺乏有效验证手段。

因此本论文拟针对当前研究的不足，将工作分为两大部分：针对上述前三条缺点，我们将在本文中提出一种新的基于操作语义的弱内存模型，它可以很好的解决这三条缺陷。而针对于第四条缺点，我们将在本文中提出一种新的适用于在 TSO 弱内存模型（具体模型的背景介绍请参见小节 2.2.1）上进行验证的程序逻辑。在本节的剩下部分，我们将分为两个小节分别介绍这两项工作：

1.1　弱内存模型的设计

已经由很多关于计算机体系结构方面的内存模型被人们提出用于允许在硬件上进行各种优化 [1]。而对于编程语言，它们的内存模型可能更加复杂，因为它们需要同时反映出编译器和硬件的优化。

设计弱内存模型的关键挑战与我们的解决方案。 我们认为，一个好的弱内存模型需要满足下面三点，这同时也是这一部分工作所面临的挑战：

- 模型对于程序员来说是有用的。这意味着，它必须满足 DRF-Guarantee 性质，这个性质说的是没有数据竞争（Data-Race-Freedom, 简称 DRF）的程序在该弱内存模型下拥有和在顺序一致性模型下相同的行为。这是保证，程序员能够基于该模型写出行为满足其预期的正确同步程序的前提。

- 这个弱内存模型约束不能太强，不然就会阻止许多重要的优化技术的在其上的实现，特别是那些已经在现存编译器中重度使用的优化更需要模型能够支持。我们可以这么说，模型约束越弱，则其能允许的优化会越多，但是同时模型的约束又不能太弱，不然就会违背第一条性质。

- 根据上面两条，理想状态下，弱内存模型需要允许任何有数据竞争程序所产生的行为，同时又要保证 DRF 程序的顺序一致性行为。然而，对于

类 Java 的类型安全语言，我们不仅仅要求这两点，还需要要求对于有数据竞争的程序，它仍然是安全。比如说，程序在弱模型下，应当不能产生 out-of-thin-air 这样凭空出现的值的行为。

事实上，要设一个内存模型同时满足上面三点要求是相当有挑战性的。

比如已存在的对程序语言构建内存模型的工作中最广为人知的是 Java 内存模型（JMM）, 这个模型已经被 Java 规范第三版第 17.4 节引入 [11]。JMM 使用了 happens-before 内存模型 (HMM) 作为 JMM 的基础。原始版本的 HMM 非常简单，并且约束也相当少，同时比起顺序一致性模型，它能够支持很多更多优化算法的实现，即它很好的满足了上面三条约束中的第二条。但是，它的因果循环（我们将在小节 2.2.2中更详细的介绍）会导致 out-of-thin-air 凭空出现的值的产生，并且导致违背了类型安全的需求。HMM 同时还违背了 DRF-Guarantee 性质。为了避免上述的原因，JMM 不得不引入 9 条公理去约束规范什么是合法的程序执行行为 [12]。这九条晦涩难懂的公理成了 JMM 中最难理解的部分。同时，由于 JMM 的模型定义不是自然生成的 (generative nature) 的，这使得程序和程序的执行轨迹之间的关联难以推理，意味着我们难以仅仅通过静态的分析程序代码来推导在这个模型下程序的行为。JMM 之后有一系列的其他工作来完善这个模型，它们同时也指出了原始的 JMM 所存在的不足之处，即 JMM 禁止了一些本应当被允许的程序行为 [13], 以及 JMM 有许多违反直观认识的特性 [14]。

针对上述的问题，我们在第三章中提出一种 HMM 的变种——OHMM (Operational Happens-Before Memory Model)。顾名思义，这是一种基于操作语义定义的内存模型。

具体来说，我们设计了一组操作语义的规则来定义 OHMM，即通过对一个简单语言赋予具体的操作语义，并通过抽象机上的状态转移来模拟 HMM 的行为。为了满足上面的三条约束，我们尽可能的简化规则。但是同时，又要保证这些规则能够模拟大部分经过不同的优化后的程序行为。由于 OHMM 所允许的程序行为是通过操作语义自然生成的，所以它自然而然的避免了所谓的凭空出现（out-of-thin-air）的程序行为。

OHMM 主要使用了三种构件（三套规则）来模拟程序的弱行为：（1）事件缓存。我们把每个对内存的读写操作看成是一个事件，事件发出的顺序是按成程序顺序由线程发出，然而在我们的模型中添加了事件缓存规则之后，OHMM 允许事件并不是立即访问内存，而是先放到事件缓存中，在之后的某个时间点再执行。这样，事件的执行顺序在不违反数据依赖性的前提下，能够做到调序执行，这将让我们的模型支持很大一部分调序优化。(2) 基于历史记录的内存结构。在 OHMM 中，每个内存单元都可以视为一组写操作队列，这组队列记录了所有对该块内存单元的写操作。通过历史记录，能够让 OHMM 上的读操作在符合一定关系的前提下能够在历史记录同时看到若干个写操作，相比于顺序模型中读操作只能看到最近的一次写操作，更加的弱化了模型的约束。(3) 重放机

制。我们允许程序中某些符合一定条件的指令在程序执行的过程中能够多次执行，来模拟现实世界中编译器和处理器优化中的投机执行。

总体来说，我们的模型对于无锁程序的约束会比 Java 内存模型（JMM）更加弱一些，因此在我们的模型中，我们能够允许更多优化算法的实现。同时，在 OHMM 上，一些程序的行为在直观上会比 JMM 更加自然。许多在 JMM 上可能出现但是明显违反直观认识的程序行为，在我们的模型上就不再合法。我们希望可以 OHMM 可以成为可供类 Java 语言选择的一种新内存模型。

我们想在这边强调一点，OHMM 是一个新的内存模型，如同我们上面提到的那样，它并没有完全和 JMM 的行为保持一致。为了集中重点，我们的模型将基于一个非常简单的程序语言来介绍，它只包括最简单的内存读写操作，同步操作以及条件分支和循环语句。我们去掉了很多其他语言特性而只保留最基本的框架，比如对象初始化，常量域以及 I/O。因此尽管我们希望我们的想法可以作为类 Java 语言的下一代内存模型，但是目前这个模型还远远未达到可以完全取代现有的 JMM 的状态。

1.2　运用于弱内存模型的程序逻辑设计

许多细粒度的并发算法允许**不使用**同步原语来同时访问共享内存。经常这类算法的设计比较巧妙，同时也让算法的正确性并不是那么直观，而形式化验证这一类算法更是具有挑战的任务。尽管已经有各种各样的程序逻辑 [5–8] 被提出来证明细粒度的并发算法，但是他们的绝大部分都基于顺序一致性模型的假设 [3]（当然，也有例外，比如 [15, 16]）, 即他们假设，程序是允许在顺序一致性模型中，不会有因为语句调序，投机预测等而产生的弱行为。而我们前面提到，现实世界中绝大部分的硬件和编译器，都是基于弱内存模型的，因此这些基于顺序一致性模型的逻辑，对于现实世界中那些运行于弱内存模型之上的程序，并不能给出任何正确性的保证。

绝大部分弱内存模型允许调换内存访问的顺序来适应模拟编译器或者处理器中的优化。对于那些有数据竞争的程序，这样的调序会比在顺序一致性模型下按照程序顺序执行时产生更多的行为。所以我们如何在弱内存模型下证明这些有数据竞争的程序呢？

在第五章中，我们提出了一个新的程序逻辑 LRGTSO 用来验证在 TSO(Total Store Order) 内存模型下的并发算法。在现实世界里实际运用于处理器和编程语言的弱内存模型中，TSO 是其中最简单也是最广泛运用的模型。它不仅仅在 X86 和 SPARC-TSO 处理器族中得到运用，并且在一些高级语言中也正在被提案作为其内存模型的基础 [17, 18]。程序在 TSO 模型上所允许的行为是前面一节中提到的 OHMM 的一个子集，我们将在第四章中证明这一点。简单来说，我们可以把 TSO 模型想象成是这么一个抽象机：所有线程拥有同一个共享内存；每个线程都有自己局部的先入先出的缓存队列；对所有的非原子写操作在写入内

存前都先放入写缓存中，然后在将来的某个时候再从其中出队并写入内存；线程所有的读操作则不需要经过缓存，而是立即执行。并且，读操作会首先尝试去缓存中寻找对同一个内存位置最新的 (即最后放入缓存的) 写操作，如果有则读取其的值，如果缓存中没有任何对该位置的写操作，则读操作从内存中直接读取。具体的定义参见第小节 2.2.1。在这一部分工作中，我们使用基于写缓存的操作语义来描述 TSO 模型 [19], 并且对冯新宇的 LRG [7] 进行了扩展，加入了描述写缓存的逻辑规则和断言。

设计 TSO 程序逻辑的关键挑战与我们的解决方案。 在传统的用于顺序一致性模型的逻辑中，如果没有其他线程的影响，断言是稳定的（稳定的意思是说，如果某个状态满足断言，那么状态在经过环境影响而产生变化到达的新的状态，仍然满足断言）。然而，在 TSO 弱内存模型下，这种稳定性会由于线程自身的写缓存非确定性地往内存中写值改变内存状态而破坏。非确定性的意思是，当缓存中存在写操作即不为空的时候，线程在任意时刻 (不确定的时刻)，都可能会将写缓存中的写操作出队写入内存。举个例子，假设我们有 Hoare 风格的三元组 $\{P\}x := 1\{Q\}$。这个三元组的含义是指，当机器状态满足 P 的时候，成功执行完 $x := 1$ 这条语句后，机器状态应当满足 Q。我们知道，在 TSO 模型下，执行 $x := 1$ 这条语句会先把这个写操作放入缓存中。因此 Q 说的是写缓存中有若干写操作，并且最后一个写操作是对 x 写入 1。但是，这样的断言 Q 是不稳定的，因为那些在写缓存中的写操作在这条赋值语句做完以后的任何时刻都可能会被线程从缓存队列中取出并写入内存，那么所到达的新的状态就不再满足 Q 了。因此我们需要去设计一个缓存稳定的断言来解决这样的问题。

另外一个挑战来自于如何去处理内存所有权转移。近年来提出的研究并发算法的逻辑系统中，很多都会在逻辑上将整个内存分为局部和共享部分，以用来支持局部推理。并且，通过一些规则，内存单元可以自由在共享和局部之间转移，我们称之为内存所有权转移。我们设计 TSO 逻辑的思路也是想让我们的逻辑支持局部推理，所以我们也需要定义内存所有权转移在什么时候能够发生。在 [7, 8] 的工作中，内存所有权转移会在执行完一条原子写操作后发生。然而，在 TSO 中我们需要面临的问题是，对于那些非原子的写操作，即那些会被放入写缓存的写操作，我们是否应该允许内存所有权转移？如果允许，应该如何设计？

第一个问题的答案是肯定的，因为我们需要去验证各种优化算法在 TSO 下的正确性，比如在 Linux 中用到的经过优化的 spin-lock 算法。在这个算法中，释放锁的时候只是使用一个普通的非原子的写指令（而不是用加了 Lock'd 前缀的原子写指令）将锁资源释放。在 TSO 模型下，尽管这个非原子的写操作已经执行，但只是被放入写缓存中，所以直到它从缓存中写入内存前，对于其他线程来说锁资源都还是未被释放的。而根据我们对程序和 TSO 模型的分析，我们知

道这样的优化在 TSO 下面仍然是可以保证互斥访问的，即 spin-lock 算法仍然是正确的。而如何严格地证明这一算法，我们就得允许内存所有权转移来方便我们的证明。在第五中节 5.4中我们使用了我们的逻辑来证明这一算法在 TSO 模型上的正确性。

对于第二个问题，我们需要去决定内存所有权转移是发生在非原子写操作放入写缓存的时候，还是发生在从写缓存写入内存的时候。在实现中，我们选择了后者，和 LRG 一样，我们通过一个全局的不变量 I 来指导内存所有权转移。

我们还观察到这么一个事实，即我们只能允许非原子写操作引起的内存所有权转移，只能把局部内存单元转移到共享内存中，而反方向是不行的。因为如果我们允许某个线程的非原子写操作引起的内存所有权转移能从共享资源转移到局部的话，那么由于这个线程看到这个非原子写操作的时间要早于其他线程（因为非原子写操作先被放入写缓存中），那么这个线程对共享资源的观察可能就会迥异与其他线程，这样的不一致性会导致数据竞争。

我们知道，TSO 模型只有一整块共享的内存，而我们的逻辑中又人为的将内存划分为共享和局部。因此为了让我们这一部分的工作解释起来更加清晰，我们对 TSO 模型进行了再一次的抽象，使之支持共享内存和局部内存的分离以及允许共享内存和局部内存之间的所有权转移。我们称新的内存模型为 ATSO。更加重要的是，由于在新的内存模型上，我们知道局部单元对其他线程不可见，因此，对于局部内存的写操作可以直接写入内存当中而不需要放入写缓存。这将可以使我们的逻辑更加简化，并且更好的支持局部推理（节 5.2和节 5.3）。我们在第五章中的节 5.5将证明，TSO 模型是 ATSO 模型的一个精化 (1)，即程序在 TSO 模型下发生的行为是在 ATSO 行为的子集。然后，我们证明了我们逻辑在 ATSO 上的可靠性 (2)，结合 (1)(2) 我们可以知道我们的逻辑在 TSO 上可靠性的成立。同时，我们还在节 5.4中运用我们的逻辑证明了一些很有代表性的程序在 TSO 模型下的正确性。

本文主要贡献。 综上两部分的工作，本文主要在弱内存模型及相关方面进行了研究，主要贡献点在于：

- 对于弱内存模型设计上的贡献：我们设计了一个新的内存模型 OHMM。
  - OHMM 使用了操作语义的方式来定义 happens-before 关系 (通常通过 axiomatic 的方式定义)，这使得我们的模型天然的避免了 HMM 中会出现的 out-of-thin-air 行为，同时我们也不再需要像 JMM 那样引入复杂公理约束来避免上述的行为。
  - 设计了重放机制来重复执行指令，这允许 OHMM 能够模拟现实世界中的投机执行行为。我们还通过一些例子将我们的模型与 JMM 进行

比较，显示了程序行为在我们的模型上比起在 JMM 上更加的符合直观认识。

– 我们还证明了，TSO 模型所允许的程序行为是我们的模型 OHMM 的一个子集。这说明 OHMM 的约束足够的弱，只要往其中加入额外的约束，就能够良好的模拟已经存在的并被广泛使用的弱内存模型。

• 对于弱内存模型上程序逻辑的贡献：我们设计了一个适用于 TSO 模型上程序验证的程序逻辑。

– 我们提供了一组缓存断言来描述 TSO 中的写缓存。为了让我们的断言语言更加简介且具有表达性，我们设计了断言稳定的 (buffer-stable) 的断言语义，所以断言的正确性和由于不确定地执行写缓存导致的状态变换是独立的，即如果某一个机器状态满足我们的断言，那么从这个机器状态开始，通过执行写缓存中的写操作而导致的状态变换所到达的新的机器状态，仍然满足断言。

– 我们的逻辑是作为 LRG 逻辑的扩展，并且支持 TSO 的局部推理。并且，它同样允许在共享内存和局部内存之间的所有权转移 —只有当同步语句执行的时（和 LRG 一样）或者写缓存对内存进行写操作时发生。如我们在节 5.4中会展示的那样，后者对于证明 Optimized Lock Implementation 算法在 TSO 下的正确性至关重要。

– 我们的工作中还展现了将局部内存和共享内存区分开的重要性。这不仅可以让我们支持局部推理，还能简化我们在 TSO 下的程序验证。我们知道，原生的 TSO 模型是没有区分共享内存和局部内存的，而在我们在具体实现中，将原生的 TSO 模型上又抽象出一层新的模型。在这个新的模型中，我们再区分出共享内存和局部内存，并且对局部内存的写操作并不通过写缓存而是直接写入内存。在这个新模型上，我们开发出我们的程序逻辑来验证。同时，我们还将证明，新的模型和原生 TSO 的等价性，这样，只要是我们逻辑验证过的程序，那么它的行为在新的模型上与在 TSO 上是一样的。

– 我们使用这个逻辑证明了一些具有代表性的并发算法在 TSO 模型上正确性，包括 Peterson’s Lock 算法，Simpson’s Four Slot 算法，Concurrent GCD 算法以及 Optimzed Implementation of Locks 算法。

本文结构与组织。 我们在第三章中详细介绍在前面节 1.1中提出的弱内存模型 OHMM；并在第五章中详细介绍在前面节 1.2中提出的适用于 TSO 弱内存模型的程序逻辑；本文其余章节中，第二章是技术背景介绍；第四章节证明 TSO 弱内存模型是 OHMM 模型程序行为的一个子集，它可以视为第三章和第五章的

过渡章节，更重要的是它说明了 OHMM 的适用性；第六章是全文的总结，而附录A,B,C,D是前面章节中提出的主要定理和推论的证明部分。

第二章　背景知识介绍

我们在本章中将对一些对理解本文内容必须的基础背景知识进行简要的介绍。熟悉这些知识的读者可以直接跳到第三章开始阅读。

2.1　顺序一致性模型

在单核 (串行) 环境下，绝大部分的编译器和硬件表现出来的行为都能够使用顺序语义来描述其内存操作。顺序语义允许程序员假设所有的内存操作都会按照程序顺序逐个执行。因此，对于读操作来说，程序员可以期待其总是返回对所读位置的，发生在读操作前面，并且按照程序顺序的最后一个写操作的值。在单核环境下，这样的语义可以被底层的优化很有效的支持。比如，底层优化只需要保证数据依赖型和控制流依赖型在优化前后维持不变（比如对于同一个内存位置的操作总是按照程序顺序的先后执行等），就可以自由的对那些互相之间没有依赖性的对不同内存位置的操作进行调序，而整个系统所表现出来的行为仍然是遵守顺序语义的。单核上的顺序语义不仅能够提供给程序员一个直观而且简单的模型，同时又允许底层的实现进行有效的优化。

到了并发环境开始流行的时候，最简单的内存模型也是基于串行上的顺序语义扩展而来，Lamport 在 [3] 中对其进行的形式化描述，并将其称为**顺序一致性模型 (SC 模型)**。我们把顺序一致性模型表现出来的行为称之为顺序一致性：

定义 2.1.1 (顺序一致性)**.** *“我们称某个多核并发环境是顺序一致的，当且仅当对运行于其上的任意执行所表现出来的行为，与以下行为一致: 所有核的内存操作，都是按照一定的顺序逐个执行，并且对于每个核来说，它的所有操作执行的顺序是按照它所运行的程序所指定的语句顺序来执行的。” By Adve, 1996 [1]*

顺序一致性模型让程序员有了一个简单而直观的模型，如图 2.1中所示，我们可以这么想象：顺序一致性模型由一个全局共享的内存单元和若干个线程组成。在每一个运算周期开始的时候，系统会随机不确定性地将某一个线程和内存连接到一起。而连接到内存上的线程可以按照程序指定的内存访问顺序来进行内存操作，然后进入到下一个运算周期，周而复始，直到所有代码执行完毕。

然而，和单线程环境下顺序语义能够允许许多有效的优化不同的是，顺序一致性模型几乎禁止了所有的调序优化以及其他很多优化的实现。即在多核环境下，即使某个线程的两条内存操作之间没有数据依赖性，如果我们把它们的执行顺序调换，那么程序所表现出来的行为就违背了顺序一致性。我们在第一章中的例1.0.1就很好地说明了这一点。所以顺序一致性模型尽管简单，但是由于实现其需要昂贵的代价，很少在现实世界的并发系统中采用（一些对性能要求并没有苛刻要求的嵌入式系统可能会使用这样的模型）。

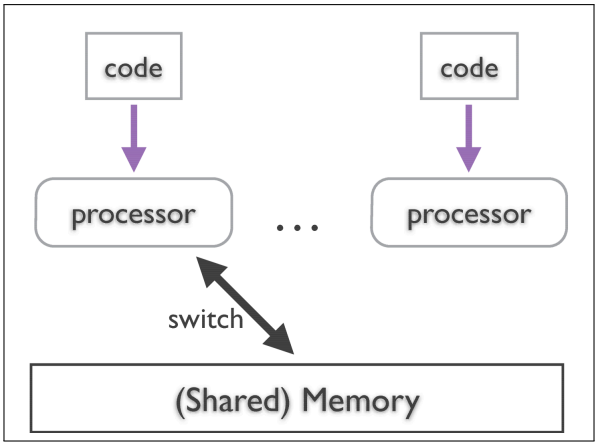


图 2.1: 顺序一致性模型示意图

2.2　弱内存模型

为了克服顺序一致性模型实现代价太高的缺陷，人们开始设计一些对内存操作顺序要求并没有那么严格的内存模型，我们称之为弱内存模型。这里的“弱” 指的是内存模型对内存操作的顺序的约束，弱于顺序一致性模型，由此显而易见，弱内存模型的表达力以及所能允许的优化实现，会比顺序一致性模型强很多。据我们所知，已经有非常多的弱内存模型被人们提出，并且不仅仅是在学术界内，在工业界，弱内存模型也得到了广泛的运用。我们准备在这一节中，简要介绍几种被广泛使用的弱内存模型，作为理解本文余下章节中我们工作的技术背景知识。

2.2.1　TSO 弱内存模型

我们在第一章中说过，弱内存模型中在工业界最广泛使用的是 TSO 模型，在 X86 和 SPARC 以及一些高级语言中都有用来表达系统的行为语义。与上一节中讲到的顺序一致性模型对所有内存操作（写和读）强制了全序关系不同，TSO(Total Store Order) 顾名思义，只对所有写操作上强制了全序关系。这意味着每个线程所观察到内存上的所有写操作顺序都是同样的顺序，并且同时还允许读操作和它**之前**的写操作之间调换顺序。和顺序一致性模型一样，我们同样可以通过一个简单的机器模型来理解 TSO 的行为。如图 2.2中所示，我们可以想象 TSO 模型由一个共享内存和一组线程组成。每个线程拥有自己局部的寄存器和写缓存。每个缓存都可以视为一个先入先出队列，线程发出的每个写操作都会首先放进自己的写缓存当中，并且在将来某个时刻（不确定性的）出队并且写入内存当中。而线程发出的读操作是立即执行（没有放入缓存），并且首先会先尝试从缓存中寻找**最新**的写操作（即缓存中所有对当前读取的内存位置的写操作中，最后放入缓存的那一个）。如果在缓存中没找到任何的对读取单元的写操作，则读操作会直接从内存中读取。

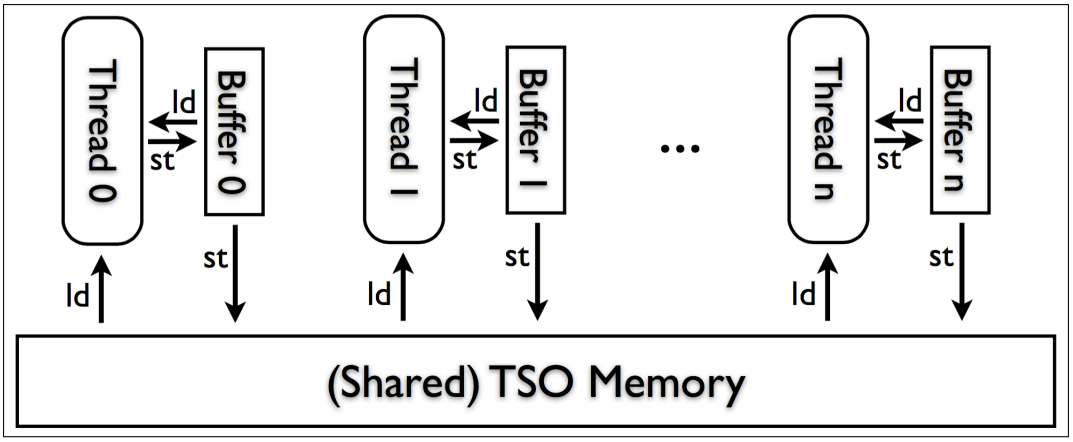


图 2.2: TSO 抽象机

例 2.2.1. TSO 执行例子:

$$\textit{Initially } x = y = 0.$$

$$\begin{array}{llll} 1: & x := 1; & 3: & y := 1; \\ 2: & r_1 := y; & 4: & r_2 := x; \end{array}$$

TSO Allowed: $r_1 = r_2 = 0$

$$\begin{array}{llll} 1: & x := 1; & 1: & r_1 := y; \\ 2: & y := 1; & 2: & r_2 := x \end{array}$$

TSO Forbidden: $r_1 = 1 \wedge r_2 = 0$

我们用例2.2.1来加深读者对 TSO 模型的理解。在例2.2.1左边的程序中，在 TSO 模型上，两个读操作能够同时读到 0。因为在 TSO 模型中，两个线程可以分别先把对 x 的写操作和对 y 的写操作放入各自的缓存中，然后执行各自的读操作，分别从内存中读到初始值 0（因为线程之间看不到对方的缓存，只能看到共享内存上的初始值）。而对于例2.2.1右边的程序中，TSO 是禁止这样的行为的，因为 TSO 要保证所有线程看到读操作写入内存的顺序一致。

我们一般使用操作语义来形式化定义 TSO 内存模型，关于 TSO 的形式化的操作语义定义，见图 5.3。

2.2.2　Happens-Before 弱内存模型

我们在这一小节中将介绍 Happens-before 弱内存模型 (HMM), HMM 也是一种被广泛使用的内存模型，包括 Java 和 C11 在内 [11, 20–24]，都使用了 HMM 的模型作为他们语言内存模型的基础。和 TSO 以及顺序一致性模型使用操作语义定义不一样的是，HMM 一般用公理语义进行定义（在我们的工作中，我们的模型是基于 HMM 的，同时我们使用了操作语义来定义，避免了使用公理语义定义 HMM 时会出现的因果循环链的问题），公理语义 (aximoatic) 通过描述两个内存操作之间的关系来约束整个模型的行为。这两种定义方法，各有利弊. 使用公理语义定义的内存模型 (axiomatic memory model) 的缺点是它们仅仅定义了什么是可以接受（合法的）执行。这在动态测试中很有用，但是却难以静态分析程序行为。换句话说，如果我们已知了某个程序行为，我们可以很容易通过

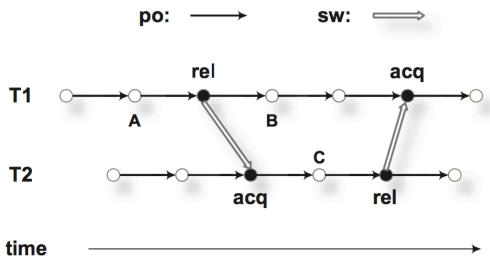


图 2.3: Happens-Before Order

axiomatic memory model 来判断其在该内存模型下是否合法，但是如果我们不知道程序行为，我们则很难去仅仅通过静态分析代码就知道程序行为到底是怎么样的。而使用操作语义定义的内存模型则正好相反。它们可以让程序员通过内存模型下静态分析程序是如何一步一步运行，而最终将产生什么样的行为，但是当程序比较复杂的时候，却很难对一个给定的程序行为判断其是否符合内存模型。

在 HMM 中，一个程序执行的过程，可以被看做是一组内存访问事件和在这些事件上的关系组成的。我们这边说的关系，包括程序顺序和同步顺序 (synchronization order)。程序顺序 $\xrightarrow{\text{po}}$ 是在来自于**同一个线程**的事件上的**全序**关系。它反映了线程是如何根据程序代码按序产生各种内存访问事件的。而同步顺序 $\xrightarrow{\text{so}}$ 则是在来自于**所有线程**的同步事件上的**全序**关系。同步事件包括申请/释放锁资源，以及对同步变量（在 Java 中，即 volatile variables）的读写。$\xrightarrow{\text{so}}$ 需要和 $\xrightarrow{\text{po}}$ 一致，即从同一个线程发出的同步事件在 $\xrightarrow{\text{so}}$ 中的先后关系必须和它们在 $\xrightarrow{\text{po}}$ 中的先后顺序一致。

同时，HMM 还定义了一种叫做 sychronization-with 的关系 $\xrightarrow{sw}$, 这种关系产生于释放锁资源的同步事件以及之后申请锁资源事件之间（所谓的之后，是根据 $\xrightarrow{\text{so}}$ 所定的顺序指定的先后）之间; 或者产生于对一个 volatile 变量进行写入以及之后对该变量的读操作之间。因此我们知道 $\xrightarrow{sw}$ 是偏序关系，并且是 $\xrightarrow{\text{so}}$ 关系的子集。我们在图 2.3中展示了 $\xrightarrow{\text{po}}$ 关系和 $\xrightarrow{sw}$ 的示意图。

最后, HMM 的核心 happens-before 关系 $\xrightarrow{\text{hb}}$ 定义为 $\xrightarrow{\text{po}}$ 和 $\xrightarrow{sw}$ 的传递闭包。如图 2.3中，我们有 A $\xrightarrow{\text{hb}}$ B 和 A $\xrightarrow{\text{hb}}$ C, 但是 B $\xrightarrow{\text{hb}}$ C 或 C $\xrightarrow{\text{hb}}$ B 均不成立。有了 happens-befoe 关系后，我们就能定义出 HMM。同时，我们也能利用 happens-before 关系在顺序一致性模型上定于无数据竞争的概念。即对于某个程序，如果它的所有符合顺序一致的执行中，它的任意读操作能且只能读到在 happens-before 关系上离它最近的写操作，那么我们就称该程序是没有数据竞争 (Data-Race-Freedom, DRF) 的。更加形式化的定义我们将在第三章中给出。在设计弱内存模型中，一个好的弱内存模型首先应该满足，对于那些 DRF 的程序，它们在该模型下的行为，应当和在顺序一致性模型一致，我们称这样的性质为

DRF-Guarantee。有了 DRF-Guarantee 才能保证，当程序员写出一个正确同步的程序，不管底层怎么优化，程序的行为总是符合程序员的预期（即顺序一致的）。

在 HMM 中，一个读操作 r 可以看到在 happens-before 关系上离它最近的写操作 w, 即 $w \xrightarrow{\mathrm{hb}} \mathrm{r}$ 或者 $\neg\exists w'. w \xrightarrow{\mathrm{hb}} w' \xrightarrow{\mathrm{hb}} \mathrm{r}$，也可以读到任何与它之间没有 happens-before 关系的写操作，即 $\neg(w \xrightarrow{\mathrm{hb}} \mathrm{r} \vee \mathrm{r} \xrightarrow{\mathrm{hb}} w)$。

HMM 是一个对内存读写操作约束相当弱的内存模型。例2.2.1中左右两个程序的行为在 HMM 下都是合理的。

HMM 所存在的问题。 HMM 存在的最严重的问题就是因果循环问题。Happens-before 关系最早是被 Lamport [25] 提出用来描述在分布式系统上基于消息传递的动作之间的因果关系。对于那些有数据竞争的程序，在一个从某个线程发出的写操作和在后续的其他线程发出的读操作之间，happens-before 关系会对这两个操作给出错误的因果关系。下面这个例2.2.2中（这个例子来自于 Manson 等人在 [12] 中的工作）显示了这个问题：

例 2.2.2. HMM 因果循环问题: *Initially* $x = y = 0$.

$$
\begin{array}{l||l}
1: \ r_1 := x; & 4: \ r_2 := y; \\
2: \ \textbf{\textit{if}}\,(r_1 \neq 0) & 5: \ \textbf{\textit{if}}\,(r_2 \neq 0) \\
3: \ \quad y := 42; & 6: \ \quad x := 42;
\end{array}
$$

Result: $r_1 = r_2 = 42$*?*

$$
\begin{array}{l||l}
1: \ r_1 := x; & 3: \ r_2 := y; \\
2: \ y := r_1; & 4: \ x := r_2;
\end{array}
$$

Result: $r_1 = r_2 = 42$*?*

例子中左右两个结果在 HMM 中都是可能发生的。首先我们注意到左边的程序事实上是 DRF 的，而 HMM 允许这样的程序行为发生，说明 HMM 本身并没有 DRF-Guarantee 的性质。左边的程序在顺序一致性模型中，程序左边的线程不会访问 y, 而右边的线程不会访问 x, 因为它们都不会进入到 IF 语句的分支中去。然而，在 HMM 中，我们可以通过一个自我证成的 (self-justifying) 的因果循环来得到这样的行为: 我们先投机预测，第三行可以被执行，所以第四行将读到 42，因此我们可以执行第 6 行，然后我们执行第 1 行也读到 42，接下来进入到第二行的条件表达式判断，我们发现条件表达式为真，因此我们的确可以执行第三行，符合了我们之前的投机预测。因此这样的行为在 HMM 下是合法的，这就形成了一个因果循环链，显然和真实世界中的行为不一致。而例子中右边的程序显示 HMM 会允许凭空出现的值 (out-of-thin-air) 的读操作发生。我们注意到 42 这个值是事实上在程序中都没有出现，它是一个凭空出现的值。然而，我们同样可以用投机预测来构造一个满足这样的读操作的因果循环链并且让它在 HMM 下是合法的。我们首先预测 r_1 会读到 42（或是随便其任何数），然后对 y 写入 42，接着让 r_2 读到 42 并对 x 写入 42，这是我们发现 r_1 的确能读到 42，符合了投机预测。

为了解决这些问题，Java 内存模型 (JMM) 在 HMM 之上引入了因果约束来定义什么是合法的执行 [12, 20]，同时这个模型也被加入到官方的 Java 规范中第十七章作为描述 Java 语言的内存模型 [11]。尽管得到了广泛的运用，但是被

许多其他研究者认为，这个模型为了限制 HMM 的因果循环，引入了太多复杂晦涩的公理约束，使得这个模型难以直观理解 [21, 26], 并且这个模型也不是完全令人满意的，还存在着一些小 BUG [13, 14]。我们将在第三章的节 3.3中，将 JMM 和我们的模型作为对比，通过一些例子更详尽的阐述这一点。

2.3　程序逻辑

我们知道，程序的形式化验证 (Formal Verification) 是证明程序正确性的一种方法（其他方法还有如模型检测等）。而形式化验证是通过由一组逻辑规则构成的逻辑系统来严格地对程序进行静态推理实现的。我们把这样的逻辑系统称之为程序逻辑。从 Hoare 在 1969 年提出 Hoare 逻辑以来 [27]，已经有各种各样的程序逻辑被提出用来证明程序正确性，其中最广泛使用的有分离逻辑 [28, 29]。而随着多核环境的流行，人们开始关心细粒度的并发算法的正确性。这些算法能够在**不使用**同步原语的情况下实现共享内存的同时访问。尽管已经有很多逻辑被提出来用于证明这些细粒度的同步程序 [5–8]，但是它们中的大部分并不能对现实世界中那些运行在基于弱内存模型假设的硬件和编译器环境下的程序实现给出正确性的保证。所以研究者们开始把目光转向能够适用于弱内存模型上的程序逻辑，比如 [15, 16]。我们在第五章节中也提出了一种新的适用于 TSO 弱内存模型的程序逻辑。由于我们的逻辑是基于冯新宇 LRG [7] 工作的扩展，增加了缓存断言使之可以适用于 TSO 模型，而 [7] 的工作又是由分离逻辑 [28] 和 Rely/Guarantee 推理 [6] 的结合而来，因此，我们在这边将简要介绍这三者，以作为理解我们逻辑的技术背景。

Rely/Guarantee 推理。 Rely/Guarantee 推理是用于共享内存的并发验证算法，它由 Jones 在 [6] 中提出。Rely/Guarantee 范式由四元组组成 $(\mathsf{p}, \mathsf{R}, \mathsf{G}, \mathsf{q})$。其中 p 和 q 是 Hoare 风格的前后条件，而 Rely 依赖条件 R 指定了线程对其他线程所造成环境状态转移的期待，而 Guarantee 保证条件 G 指明了线程自身对环境所能产生的影响。

我们说一个程序满足 Rely/Guarantee 范式，即 $C\ \mathsf{sat}\ (\mathsf{p}, \mathsf{R}, \mathsf{G}, \mathsf{q})$, 是指当给定一个初始状态满足前条件 p 并且环境的状态转移满足依赖条件 R, 并且由程序 C 中的每一步原子指令执行所造成的状态转换满足保证条件 G 的话，则在 C 能够顺利执行终止的前提下，最终状态满足后条件 q。

另外，Rely/Guarantee 范式还有一个额外的条件需要满足：前后条件 p 和 q 必须在依赖条件 R 下面是稳定的，意思是说它们能够抵抗环境的影响，即我们说某个 p 在 R 下面是稳定的，当且仅当对于任何的状态 s, 如果 s 满足 p, 那么从 s 出发，由 R 影响所造成的环境状态转移，到达的任意状态 s'，我们都仍然有 s' 满足 p。

分离逻辑。 分离逻辑 (Separation Logi) [28, 29] 是 Hoare 逻辑的扩展。它能有效地推理内存别名。分离逻辑使用形如 $x \mapsto n$ 的断言表示内存中只有一块单元，由 x 指向并且保存在该单元上的值为 n. 断言语言中的分离符号 (separating conjunction) 则用来将不同的内存单元结合起来，比如 $p * q$ 指定了内存状态可以被分为两块互不相交的内存，并且分别满足 p 和 q。

分离逻辑的优点是断言可以描述内存的部分状态进行局部推理，并且通过 frame 规则来实现拓展到描述整个内存上：

$$\frac{\{p\}C\{q\}}{\{p * r\}C\{q * r\}} \qquad \text{Frame Rule}$$

上面这个推导规则说的是，如果 $\{p\}C\{q\}$ 成立（即如果内存状态满足 p，那么在 C 顺利执行终止后，内存状态将满足 q），那么我们往内存上加上任意块由 r 指定的内存，在新的内存状态下，我们会有 $\{p * r\}C\{q * r\}$ 成立。我们需要注意到 C 不会访问由 r 指定的部分内存，因此 r 在 C 执行前后都能满足。

结合前两者 SAGL [9], RGSep [8] 和 LRG [7] 的工作中都尝试着将 Rely / Guarantee 推理和分离逻辑结合在一起，用于细粒度并发验证。这些工作所提出的逻辑都能通过拓展定义分离逻辑中的分离符号来将整块内存状态显性地在逻辑上分为共享和局部区域。而 Rely/Guarantee 条件则只需要用来指定那些在共享区域上的内存状态变化，而不需要像之前一起需要指定整个内存状态。这样，可以比传统的 Rely-Guarantee 推理具有局部推理的优点，并且能大大简化证明过程。就像我们在前面所说的那样，在本文第五章中，我们继承这种结合两种逻辑系统的思路并且进行扩展来使我们的逻辑系统支持在 TSO 弱内存模型下的验证。

第三章　基于操作语义的 Happens-Before 弱内存模型

Java 内存模型 (JMM) 使用 Happens-before 内存模型 (HMM) 作为其模型的基础。尽管 HMM 自身很简单，但是允许由于程序因果循环而产生不符合直观，也应当被禁止的行为。因此 JMM 不得不引入一些复杂的公理来避免因果循环的发生——因果循环会导致荒谬的 out-of-thin-air 的读操作的发生，从而破坏了 Java 的类型安全的保证。这些引入的公理导致 JMM 过于复杂并且难以理解。同时它还有许多违反直觉的行为，比如在 Aspinall 和 Ševčík 的论文中 [14] 提出的 “ugle examples”。此外，HMM（和 JMM）仅仅规范了什么执行轨迹是可以接受的，但是无法说明这些执行轨迹是怎么被产生的。这导致难以静态的分析程序在该弱内存模型上的行为。

在本章中，我们提出一种新的内存模型 OHMM，这个模型是 HMM 的变种。OHMM 通过对一个简单语言赋予具体的操作语义，并通过它在抽象机上的程序行为来模拟 Happens-before 内存模型。由于 OHMM 所允许的程序行为是通过操作语义自然生成的，所以它自然而然的避免了所谓的凭空出现 (out-of-thin-air) 的程序行为。另外一方面，OHMM 使用一种我们称之为重放的机制来允许某条符合一定条件的指令在抽象机上能够多次执行，来模拟现实世界中编译器和处理器优化中的投机执行和优化。总的来说，我们的模型对于无锁程序的约束会比 Java 内存模型 (JMM) 更加弱一些，因此我们将会允许更多的编译器优化算法在我们的模型上能够使用。同时，在 OHMM 上，程序行为在直观上会比 JMM 更加自然。许多在 JMM 上可能出现但是明显违反直观认识的程序，在我们的模型上就不再合法。我们希望可以 OHMM 可以成为可供类 Java 语言选择的一种新内存模型。

本章的结构如下: 我们先给出模型的非形式化描述和基本设定（节 3.1）以及形式化定义的操作语义（节 3.2）。接下来我们将展示一些简单的程序并分析它在我们的模型上的程序行为，并和在 JMM 模型上的行为作出对比（节 3.3）。之后我们将给出我们模型的一项重要性质的证明，即对于那些 DRF 的程序，我们的模型能够保证，其行为和在顺序一致性模型 (SCM, Sequential Consistence Model) 上的程序行为一致（节 3.4）。然后我们在节 3.5中探讨，我们的模型能够支持哪些程序变换, 并且证明这些程序变换的准确性（JMM 并没法完全支持这些变换）。在接下来的一节中我们则将讨论一些相关工作并和他们作对比（节 3.6。最后，我们将会对我们的模型做一个总结，并且列出我们模型现在仍然存在的弱点以及未来我们应当努力的方向（节 3.7）。

$$
\begin{aligned}
(\mathit{Number})\ & \mathfrak{n} \in \mathit{Integer} \\
(\mathit{NormVar})\ & \mathfrak{x}, \mathfrak{y}, \mathfrak{z}, \ldots \\
(\mathit{VolVar})\ & v, v_1, v_2, v_3, \ldots \\
(\mathit{Lock})\ & \mathfrak{l} ::= \mathfrak{l}_0 \mid \mathfrak{l}_1 \mid \mathfrak{l}_2 \mid \ldots \\
(\mathit{Reg})\ & r ::= r_0 \mid r_1 \mid r_2 \mid \ldots \\
(\mathit{Expr})\ & E ::= r \mid \mathfrak{n} \mid op(E_1, ..., E_{\mathfrak{n}}) \\
(\mathit{Instr})\ & \mathfrak{i} ::= \mathfrak{i}_{\mathfrak{n}} \mid \mathfrak{i}_{\mathfrak{s}} \\
(\mathit{NonSyncI})\ & \mathfrak{i}_{\mathfrak{n}} ::= \mathfrak{x} := r \mid r := \mathfrak{x} \mid r := \mathsf{E} \mid \mathfrak{x} := \mathfrak{n} \\
(\mathit{SyncI})\ & \mathfrak{i}_{\mathfrak{s}} ::= v := r \mid r := v \mid v := \mathfrak{n} \mid \textbf{lock}\ \mathfrak{l} \mid \textbf{unlock}\ \mathfrak{l} \\
(\mathit{Stmts})\ & C ::= \mathfrak{i} \mid \textbf{skip} \mid C; C \mid \textbf{if}\ r\ \textbf{then}\ C\ \textbf{else}\ C \mid \textbf{while}\ r\ \textbf{do}\ C \\
(\mathit{ThrdID})\ & tid \in \mathit{Nat} \\
(\mathit{Program})\ & P ::= tid.\mathsf{C} \mid tid.\mathsf{C} \,\|\, \mathsf{P}
\end{aligned}
$$

图 3.1: OHMM 的语言文法

3.1　OHMM 非形式化的描述

在这一节中，我们将给出我们模型的半形式化的描述，包括所支持的程序语言，抽象机模型，以及代码是如何在该抽象机上执行的，由此来解释我们的模型是如何设计的以及为什么这么设计。形式化的操作语义我们将在下一节中给出 (节 3.2)。值得注意的是，我们的宏观意图是为了设计一个尽可能支持更多优化（即尽可能的弱化我们模型的约束），并且同时还能保证对于没有数据竞争的程序，其行为和顺序一致性模型上能够保持一致。除此之外，我们还希望我们的模型能够做到简单易用。

3.1.1　OHMM 语言

语言的文法如图 3.1所示。在这个简化的抽象语言中，一个程序 P 是由一个或者多个串行线程组成。每个线程拥有自己的线程号 *tid* 和代码 *C*. 单独一句代码可以是一条原始指令 $\mathfrak{i}$, 或者是 **skip**，又或者是前两者的各种组合。原始指令根据它们的作用不同，被分为两类：用来同步线程的同步指令 $\mathfrak{i}_{\mathfrak{s}}$ 和普通指令 $\mathfrak{i}_{\mathfrak{n}}$。普通指令包括访问全局的非 volatile 变量的读写指令, 以及只访问线程局部变量的指令 ($r := E$)。访问全局 volatile 变量的读写指令和请求/释放同步锁的指令则被归类为同步指令。我们使用 r 来代表寄存器，它们只在所属的线程内部可见，是线程局部变量; 用 $\mathfrak{x}$, $\mathfrak{y}$ 和 $\mathfrak{z}$ 等来表示全局的非 volatile 变量; 用 v_1 和 v_2 用来表示 volatile 变量。在我们的语言中，表达式 E 是作用在常量和寄存器变量上的数学运算。我们称 $r := E$ 是 “纯指令”，因为它没有访问任何的共享变量。

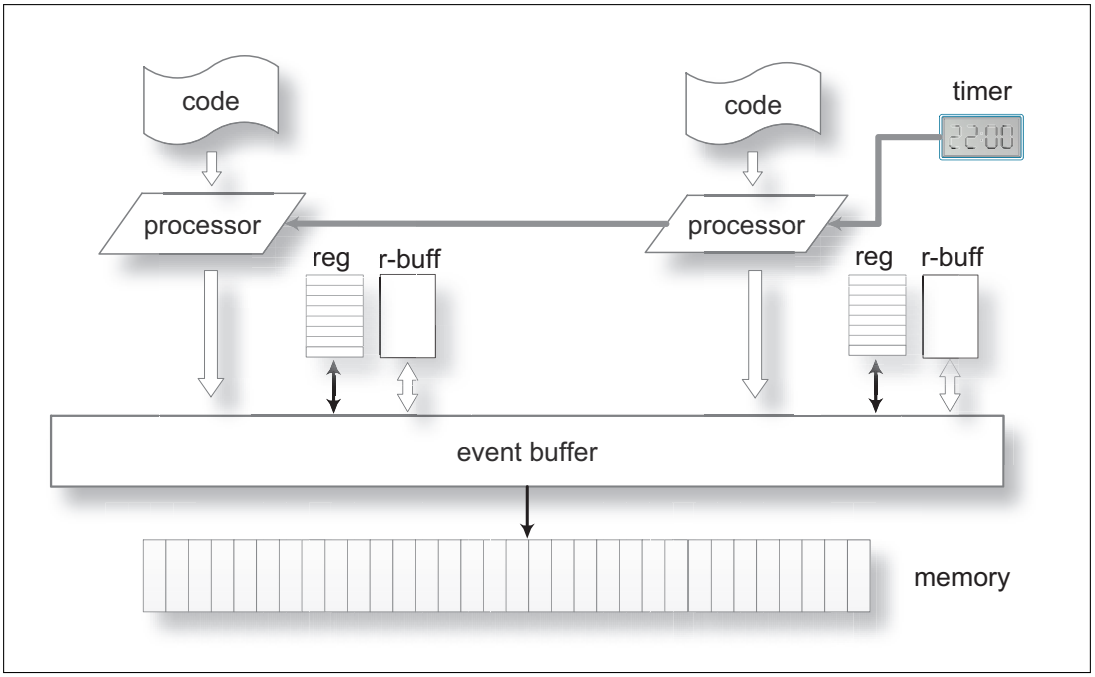


图 3.2: OHMM 抽象机设计

3.1.2　OHMM 抽象机

OHMM 的抽象机如图 3.2 所示。接下来我们会从顺序一致性抽象机（图 2.1）开始，通过往其中加入各种组件来弱化它的约束，最后使其变成我们 OHMM 的抽象机。这也是我们当初设计这样一个内存模型所经历的思考步骤。通过这样的介绍方法，读者可能能更好的了解为什么我们的模型怎么弱化约束以及为什么需要这些弱化。总的来说，我们的模型是在顺序一致性的抽象机上加入了三种构件来弱化对内存读写的顺序的约束：全局线程共享的事件缓存，基于历史记录的共享内存，以及线程局部的重放缓存。

- **事件缓存 (Event Buffer)：** 我们知道现实世界中有许多优化，是通过调整那些没有数据依赖性的指令之间的执行顺序来实现的。而顺序一致性模型的抽象机是不支持这样的优化的。为了让我们的抽象机能够模拟这种优化，我们往顺序一致性抽象机中添加事件缓存这样的构件。它的具体作用是让线程在运行代码的时候，并不是直接访问内存，而是把和代码相应的对内存的读写操作包装成一个事件 (event)，然后将这个事件放入全局的事件缓存中。这样能够使我们弱化那些没有数据依赖性的事件的执行顺序。

- **基于历史记录的共享内存 (History-Based Memory)：** 有时并发程序的行为看起来好像对于同一块内存单元来说，不同的线程读到的写操作是不同的顺序。这是因为在现实世界中，比如 CPU 中 Cache 的使用，以及各种各样的同步机制等都会导致程序行为产生这样的结果。为了模拟这类优化并进一步弱化模型，我们使用基于历史记录的共享内存。即每个存储非 volatile

$$
\begin{array}{rcll}
(\mathit{Timer}) & \mathtt{t} & \in & \mathit{Nat} \\
(\mathit{TStamp}) & \mathit{ts} & ::= & \langle \mathit{tid}, \mathtt{t} \rangle \mid \textbf{init} \\
(\mathit{Event}) & \mathtt{e} & ::= & \langle \mathit{ts}, \iota \rangle \\
(\mathit{EvtBuff}) & \mathtt{b}, \mathit{rb} & ::= & \{\mathtt{e}_0, \ldots, \mathtt{e}_n\} \\
(\mathit{RegFile}) & \mathit{rf} & ::= & \{r_0 \rightsquigarrow \mathtt{n}_0, \ldots, r_k \rightsquigarrow \mathtt{n}_k\} \\
(\mathit{ThrdQ}) & \mathit{tq} & ::= & \{\mathit{tid}_0 \rightsquigarrow (\mathit{rf}_0, \mathit{rb}_0), \ldots, \mathit{tid}_k \rightsquigarrow (\mathit{rf}_k, \mathit{rb}_k)\} \\
(\mathit{Viewed}) & \mu & ::= & \textbf{true} \mid \textbf{false} \\
(\mathit{WtOpr}) & \mathit{wv} & ::= & \langle \mathit{ts}, \mathtt{n}, \mu \rangle \\
(\mathit{SyncAct}) & \mathit{syn} & ::= & \langle \mathit{ts}, \textbf{st}, v \rangle \mid \langle \mathit{ts}, \textbf{ld}, v \rangle \mid \langle \mathit{ts}, \textbf{rel}, \mathtt{l} \rangle \mid \langle \mathit{ts}, \textbf{acq}, \mathtt{l} \rangle \\
(\mathit{HistOpr}) & \mathtt{o} & ::= & \mathit{wv} \mid \mathit{syn} \\
(\mathit{History}) & \mathtt{h} & ::= & \{\mathtt{o}_0, \ldots, \mathtt{o}_n\} \\
(\mathit{Mem}) & \mathit{m} & ::= & \{\mathtt{x} \rightsquigarrow \mathtt{h}_1, \mathtt{y} \rightsquigarrow \mathtt{h}_2, \ldots, v_1 \rightsquigarrow \mathtt{n}_1, v_2 \rightsquigarrow \mathtt{n}_2, \ldots\} \\
(\mathit{LockSet}) & \mathtt{L} & ::= & \{\mathtt{l}_0 \rightsquigarrow \mathit{tid}_0, \ldots, \mathtt{l}_k \rightsquigarrow \mathit{tid}_k)\} \\
(\mathit{State}) & \sigma & ::= & (\mathit{tq}, \mathit{m}, \mathtt{b}, \mathtt{t}, \mathtt{L})
\end{array}
$$

图 3.3: 抽象机模型定义

变量 (为了描述方便，我们在接下来的章节中，会把这类变量称之为普通变量) 的内存单元，都记录了所有对这块单元进行的写操作。因此，对于每一个读操作，在内存单元中，它都可能看到好几个不同的备选值来供其读取。

- **重放缓存 (Replay Buffer)：** 这是我们模型中最难理解的一部分内容。直观上，引入重放缓存，是为了模拟现实世界中的投机执行的优化。在下文中我们将会对其进行详细的解释。现在读者只需要知道，重放缓存允许线程重复执行某些事件。即当线程在执行某些事件以后，它可以将该事件放入重放缓存，以待将来某些时候从缓存中取出并重复执行。重放机制允许我们模拟投机执行和一些编译器通过程序分析所进行的优化。

在本节中剩下的部分，我们将会详细解释上文所列的三种构件是如何弱化模型并且同时保持 DRF-Guarantee 特性的。我们在这里想强调的是，我们的抽象机仅仅是设计用来模拟程序行为的，让我们的模型能够符合现实世界中那些并发程序所表现出来的行为。我们并不是想将这些构件实际用于现实世界的硬件或者软件优化。

事件和事件缓存。程序 P 中个每个线程都会独立在抽象机上的不同的处理核上运行。这些线程的执行顺序遵守标准的顺序的交错语义, 就像在顺序一致性模型中一样。然后，当一条指令被执行的时候，它对机器状态的影响并不是立即发生。相反的，执行该线程的处理核会发出一条相应的事件，并且把它放入全局的*事件缓存*中。如图 3.3所示，事件缓存 $\mathtt{b}$ 的类型是一组事件的集合。单个事件 e 是二元组类型 (pair)，形如 $\langle \mathit{ts}, \iota \rangle$。它由两部分组成，指令 ι 和记录该事件是由什么线程发出的以及什么事件发出的时间戳 ts。因此时间戳 ts 也是二元组

类型，它由线程 ID 和逻辑时间 t 组成。后者是一个全局的计数器，被所有处理核所共享（参见图 3.2）。当一个事件被放入缓存时，逻辑时间会自增 1 单位。下面我们将用这样的标示 *ts*.*tid* 和 *ts*.t 来分别表示 *ts* 中的第一元和第二元。我们有一个特殊的时间戳 **init**, 这个时间戳仅仅用来表示当抽象机被初始化的时候的初始时间。

有了时间戳，我们就能区分两个事件是否来自于同一个线程，并且如果它们确实来自同一个线程，我们还能区分哪个被放入事件缓存中的时间更早。我们使用 $ts < ts'$ 表示两个时间戳拥有相同的线程 ID 并且 *ts* 的逻辑时间比 ts' 的逻辑时间要小。我们同时还定义 **init** 这个特殊的时间戳比其他所有的时间戳都要小。形式化的定义如图 3.4所示。值得注意的是，对于包含非同步指令的事件来说，只有当它们来自于同一个线程，才有时间戳上的可比性，即谁比谁更早只在当它们来自同一个线程时才有定义。

线程局部数据和线程队列。每个线程都有一个线程局部可见的寄存器文件 *rf* 来指明某个寄存器（图 3.2 中的 “reg” ）当前存储的值，它是一个部分函数 (parital function), 将寄存器名字映射到整数类型上。每个线程同时还拥有一个线程局部可见的重放缓存 *rb*, 我们将会在小节 3.1.5中详细解释。线程队列 *tq* 也是一个部分函数，它将线程 ID 映射到线程的局部状态。

基于历史记录的共享内存。共享内存 *m* 也是一个部分函数，它将变量名映射到相应的值上。我们对 volatile 变量和非 volatile 变量所存储的内存单元进行不同的定义。一个 volatile 内存单元只保存了变量当前的值。而对于非 volatile 内存单元来说，我们用一个栈类型 h 来保存所有的对该单元的历史写动作。对于这种内存单元的写操作，并不会直接覆盖之前的值，而是被推入到栈顶。同时，动作推入到栈中，其作用下文将会介绍。

一个写动作是一个三元组类型 $\langle ts, \mathfrak{n}, \mu \rangle$。它记录了时间戳和写入值。而布尔类型的标记 μ 则是用来记录这个写动作所包含的值是否已经被其他线程所读取。因此，标记位的初始值为 **false**。采用这个标示是因为我们的模型支持重放机制，允许将某条事件重复执行多次。对于某一个写事件来说，在它的重复执行过程中，很有可能所写入的值各不相同 9。但是为了保证我们模型不会出现凭空出现的值，我们必须规定，当一个写事件所对应的写动作已经被其他线程所读取的以后，该写事件就不允许再重复执行。我们将会在小节 3.1.5中进行更详细的解释。同步动作 *syn* 包括请求/释放同步锁和写入/读取 volatie 变量。

整个抽象机状态 σ 由线程队列 *tq*, 共享内存 *m*，事件缓存 b, 计时器 t 和锁状态 L 组成。L 将锁资源映射到拥有该所资源的线程 ID 上。

3.1.3　事件执行顺序

暂放于事件缓存中的事件并不需要按照程序顺序按序执行。事实上，它们可以在任意时间被执行，只要以下的依赖条件能够被满足（形式化定义见图 3.4）。

$$ts_1 < ts_2 \stackrel{\text{def}}{=} ts_1 = \textbf{init} \wedge ts_2 \neq ts_1 \vee ts_1.tid = ts_2.tid \wedge ts_1.\mathrm{t} < ts_2.\mathrm{t}$$

$$UseR(E) \stackrel{\text{def}}{=} \begin{cases} \{r\} & \text{if } E = r \\ \emptyset & \text{if } E = \mathrm{n} \\ \bigcup_{\mathrm{i}\in[1..\mathrm{n}]} UseR(E_\mathrm{i}) & \text{if } E = op(E_1, \ldots, E_\mathrm{n}) \end{cases}$$

$$UseR(\iota) \stackrel{\text{def}}{=} \begin{cases} \{r\} & \text{if } \iota = (\_ := r) \\ UseR(E) & \text{if } \iota = (\_ := E) \\ \emptyset & \text{otherwise} \end{cases}$$

$$UpdR(\iota) \stackrel{\text{def}}{=} \begin{cases} \{r\} & \text{if } \iota = (r := \_) \\ \emptyset & \text{otherwise} \end{cases} \qquad UpdR(rb) \stackrel{\text{def}}{=} \bigcup_{e\in rb} UpdR(\mathrm{e}.\iota)$$

$$UseM(\iota) \stackrel{\text{def}}{=} \begin{cases} \{\mathrm{x}\} & \text{if } \iota = (r := \mathrm{x}) \\ \emptyset & \text{otherwise} \end{cases} \qquad UpdM(\iota) \stackrel{\text{def}}{=} \begin{cases} \{\mathrm{x}\} & \text{if } \iota = (\mathrm{x} := \_) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathrm{e}_1 \stackrel{\mathbf{r}}{\leftarrow} \mathrm{e}_2 &\stackrel{\text{def}}{=} \mathrm{e}_1.ts < \mathrm{e}_2.ts \wedge (UseR(\mathrm{e}_1.\iota) \cap UpdR(\mathrm{e}_2.\iota) \neq \emptyset \\ &\qquad \vee UseR(\mathrm{e}_2.\iota) \cap UpdR(\mathrm{e}_1.\iota) \neq \emptyset \\ &\qquad \vee UpdR(\mathrm{e}_1.\iota) \cap UpdR(\mathrm{e}_2.\iota) \neq \emptyset) \\ \mathrm{e}_1 \stackrel{\mathbf{b}}{\leftarrow} \mathrm{e}_2 &\stackrel{\text{def}}{=} \mathrm{e}_1.ts < \mathrm{e}_2.ts \wedge (\mathrm{e}_2.\iota = (\textbf{unlock } \_) \vee \mathrm{e}_2.\iota = (v := r) \vee \mathrm{e}_1.\iota = (r := v)) \\ \mathrm{e}_1 \stackrel{\mathbf{m}}{\leftarrow} \mathrm{e}_2 &\stackrel{\text{def}}{=} \mathrm{e}_1.ts < \mathrm{e}_2.ts \wedge UpdM(\mathrm{e}_1.\iota) \cap UseM(\mathrm{e}_2.\iota) \neq \emptyset \\ \mathrm{e}_1 \stackrel{\mathbf{s}}{\leftarrow} \mathrm{e}_2 &\stackrel{\text{def}}{=} \mathrm{e}_1.ts.\mathrm{t} < \mathrm{e}_2.ts.\mathrm{t} \wedge \mathrm{e}_1.\iota \in SyncI \wedge \mathrm{e}_2.\iota \in SyncI \\ \mathrm{e}_1 \leftarrow \mathrm{e}_2 &\stackrel{\text{def}}{=} (\mathrm{e}_1 \stackrel{\mathbf{r}}{\leftarrow} \mathrm{e}_2) \vee (\mathrm{e}_1 \stackrel{\mathbf{b}}{\leftarrow} \mathrm{e}_2) \vee (\mathrm{e}_1 \stackrel{\mathbf{m}}{\leftarrow} \mathrm{e}_2) \vee (\mathrm{e}_1 \stackrel{\mathbf{s}}{\leftarrow} \mathrm{e}_2) \\ readyR(ts, r, \mathrm{b}) &\stackrel{\text{def}}{=} \neg\exists(\mathrm{e} \in \mathrm{b}).\, \mathrm{e}.ts < ts \wedge r \in UpdR(\mathrm{e}.\iota) \end{aligned}$$

图 3.4: 事件依赖性以及辅助定义

- 寄存器依赖性 ($\mathrm{e}_1 \stackrel{\mathbf{r}}{\leftarrow} \mathrm{e}_2$)。如果事件 e_1 和 e_2 中，有一个事件读取或者更新另一个事件所更新的寄存器值，那么事件 e_2 必须在时间戳更早的事件 e_1 之后执行。在图 3.4中，我们使用 $UseR(\iota)$ and $UpdR(\iota)$ 来分别表达由指令 ι 所读取的寄存器集合和所更新的寄存器集合。

- 内存依赖性 ($\mathrm{e}_1 \stackrel{\mathbf{m}}{\leftarrow} \mathrm{e}_2$)。如果读内存事件 e_2 读取的变量是写内存事件 e_1 所更新的变量, 并且 e_1 的时间戳比 e_2 更早，那么 evt_2 必须等待比它更早的 evt_1 执行后才能执行。在 图 3.4中，我们使用 $UseM(\iota)$ 和 $UpdM(\iota)$ 来分别代表由 ι 所读取的非 volatile 变量集合和所更新的非 volatile 变量集合。

- Barriers 依赖性 ($\mathrm{e}_1 \stackrel{\mathbf{b}}{\leftarrow} \mathrm{e}_2$)。内存访问事件必须等待在它之前更早的请求锁事件和读取 volatile 变量值事件完成后才能执行。而释放同步锁事件和写入 volatile 变量事件必须等待所有在它们之前的内存访问事件完成后才能执行。

- 同步顺序 ($\mathrm{e}_1 \stackrel{\mathbf{s}}{\leftarrow} \mathrm{e}_2$)。同步事件（请求/释放同步锁，读取/写入 volatile 变量）的执行顺序必须按照它们被放入事件缓存的顺序来执行，不论它们是否来自于同一个线程与否。同时，这也解释了我们为什么在抽象机 σ 里面

需要一个全局的计时器构件 t，因为我们需要定义这么一个跨越不同线程的同步顺序。

可能内存依赖关系 $\_ \overset{\mathbf{m}}{\leftarrow} \_$ 第一眼看上去会有点奇怪，因为它并没有要求写内存事件必须等待更早的读写内存事件的完成。我们将会在下面在介绍基于历史记录的内存单元和内存访问的执行时再解释这一点。

回顾一下第一章中的例1.0.1所示的结果，通过事件缓存，我们可以在 OHMM 上得到相同的结果。我们假定在这个例子中，下面的事件是按照交错语义被封装的。

$$\langle\langle tid_1, 0\rangle, \mathrm{x} := 1\rangle, \quad \langle\langle tid_2, 1\rangle, \mathrm{y} := 1\rangle$$
$$\langle\langle tid_2, 2\rangle, \mathrm{r}_2 := \mathrm{x}\rangle, \quad \langle\langle tid_1, 3\rangle, \mathrm{r}_1 := \mathrm{y}\rangle$$

那么按照 $2-3-0-1$ 的顺序执行这些事件 (数字标示上面相应的事件的逻辑时间)。

在接下来所有的例子中，我们约定成俗一点：**抽象机内存的每个内存单元初始值都是 0，即对于所有的普通变量 x，我们有 $\langle \mathbf{init}, 0, \mathbf{true}\rangle \in \boldsymbol{m}(\mathrm{x})$。**

例 3.1.1.

$$\begin{array}{l|l|l} 1:\ \mathrm{x} := 1; & \| & 3:\ r_1 := v_1; \\ 2:\ v_1 := 1; & \| & 4:\ \mathbf{if}\ (r_1)\ \ r_2 := \mathrm{x}; \end{array}$$

对于这个例子，$r_1 = 1$ 并且 $r_2 = 0$ 的程序运行结果是不可能发生的。这是因为从标示为 *2* 的语句产生的事件 e_2 不能在从标示为 *1* 的语句产生的事件 e_1 之前执行，我们可以从 $e_1 \overset{b}{\leftarrow} e_2$ 知道这一点。类似的，标示为 *4* 的语句依赖于标示为 *3* 的语句（即其所对应的事件必须在语句 *3* 所产生的事件之后执行），因此当语句 *3* 所对应的事件读到值 *1* 的时候，语句 *1* 所对应的事件必然已经执行。这个程序事实上是一个没有数据竞争的程序。

3.1.4　历史记录和内存访问

对事件进行调序执行已经足够允许我们的模型产生许多弱行为。然而如果我们仅仅止步于此并只采用标准的内存单元模型即每个内存单元都只保存着最近一次的写操作的值的话，这样的模型的约束还是不够弱的。下面这个例子能够很好地说明这一点：

例 3.1.2 (Taken from [20].)**.**

$$\begin{array}{l|l|l} 1:\ \mathrm{x} := 1; & \| & 3:\ \mathrm{x} := 2; \\ 2:\ r_1 := \mathrm{x}; & \| & 4:\ r_2 := \mathrm{x}; \end{array}$$

Result: $r_1 = 2$ *and* $r_2 = 1$*?*

上面这个例子的所示的结果在 JMM 中是被允许的，但我们却无法通过对指令进行调序执行来产生相似的结果，因为对于左右两个线程来说，它们代码中的两条语句都是有依赖关系的（根据 $\_ \overset{\mathbf{m}}{\leftarrow} \_$ 的定义），因此我们无法调序。

下面我们将把历史记录引入模型，并且解释内存单元是如何被访问的。

$$AddSyn(m, syn) \stackrel{\text{def}}{=} \lambda \mathrm{x}. \begin{cases} \mathrm{h} \cup \{syn\} & \text{if } m(\mathrm{x}) = \mathrm{h} \\ \mathrm{n} & \text{if } m(\mathrm{x}) = \mathrm{n} \end{cases}$$

$$\mathrm{o}_1 \prec^{\text{po}} \mathrm{o}_2 \stackrel{\text{def}}{=} \mathrm{o}_1.ts < \mathrm{o}_2.ts$$

$$\mathrm{o}_1 \prec^{\text{sw}} \mathrm{o}_2 \stackrel{\text{def}}{=} \mathrm{o}_1.ts.\mathrm{t} < \mathrm{o}_2.ts.\mathrm{t} \wedge (\exists v.\, \mathrm{o}_1 = \langle \_, \mathbf{st}, v \rangle \wedge \mathrm{o}_2 = \langle \_, \mathbf{ld}, v \rangle \vee \exists \mathrm{l}.\, \mathrm{o}_1 = \langle \_, \mathbf{rel}, \mathrm{l} \rangle \wedge \mathrm{o}_2 = \langle \_, \mathbf{acq}, \mathrm{l} \rangle)$$

$$\mathrm{o}_1 \prec^{\text{hb}}_{\mathrm{h}} \mathrm{o}_2 \stackrel{\text{def}}{=} (\mathrm{o}_1, \mathrm{o}_2) \in ((\prec^{\text{po}} \cup \prec^{\text{sw}}) \cap (\mathrm{h} \times \mathrm{h}))^+$$

$$ts \prec^{\text{hb}}_{\mathrm{h}} \mathrm{o} \stackrel{\text{def}}{=} \exists \mathrm{n}, \mu.\, \langle ts, \mathrm{n}, \mu \rangle \prec^{\text{hb}}_{\mathrm{h} \cup \{\langle ts, \mathrm{n}, \mu \rangle\}} \mathrm{o}$$

$$\mathrm{o} \prec^{\text{hb}}_{\mathrm{h}} ts \stackrel{\text{def}}{=} \exists \mathrm{n}, \mu.\, \mathrm{o} \prec^{\text{hb}}_{\mathrm{h} \cup \{\langle ts, \mathrm{n}, \mu \rangle\}} \langle ts, \mathrm{n}, \mu \rangle$$

$$visible(ts, wv, \mathrm{h}) \stackrel{\text{def}}{=} (wv \prec^{\text{hb}}_{\mathrm{h}} ts \wedge \neg \exists wv'.\, wv \prec^{\text{hb}}_{\mathrm{h}} wv' \wedge wv' \prec^{\text{hb}}_{\mathrm{h}} ts) \vee \neg (wv \prec^{\text{hb}}_{\mathrm{h}} ts \vee ts \prec^{\text{hb}}_{\mathrm{h}} wv)$$

图 3.5: 更多辅助定义

普通变量的写操作。对于普通变量的写事件 $\langle ts, \mathrm{x} := r \rangle$ 来说，我们可以简单把写动作 $\langle ts, \mathrm{n}, \mathbf{false} \rangle$ 放入相应内存单元的历史记录 $(m(\mathrm{x}))$ 中。这里 n 是 r 的值（如果 r 已经 “准备” 好被看到的话, 参见图 3.4中的定义 $readyR(ts, r, \mathrm{b})$）。标示位为 **false** 意思是说这个写操作还未被其他任何线程所看到，我们下面会解释这一点。

对 volatile 变量的读写操作。对于 volatile 变量来说，我们在它的内存单元中只保存它的最新值，因为这种变量的值的更新对于所有线程来说总是同步的。即读操作会得到某个存在该单元上的值，而写操作则会直接覆盖存在单元上的值。然而，由于访问 volatile 变量属于同步操作，我们需要在所有的非 volatile 变量的历史记录中记录形如 $\langle ts, \mathbf{st}, v \rangle$ 和 $\langle ts, \mathbf{ld}, v \rangle$ 的动作（具体的形式化定义请参见图 3.5中的 $AddSyn(m, syn)$）。类似的，对于所有的请求和释放锁操作，我们也在所有非 volatile 变量的历史记录中记录相应的同步动作。

对普通变量的读操作。为了展示普通变量的读操作是如何工作的，我们首先在图 3.5中定义 Happens-before 顺序 $\prec^{\text{hb}}_{\mathrm{h}}$。它是程序顺序 $\prec^{\text{po}}$ 和 synchronizes-with 顺序 $\prec^{\text{sw}}$ 的传递闭包。值得注意的是，我们这里提到的 Happens-before 顺序，程序顺序以及 synchronizes-with 顺序和我们在第二节中介绍 HMM 时提到的同名顺序严格来讲，在形式化定义上并不是完全相同的（因为为了符合我们的模型）, 但是在直观上读者可以简单地认为两者是相同的。然后，我们重载了 $\prec^{\text{hb}}_{\mathrm{h}}$ 的定义，来标示在 h 中的某个动作 o “发生” 在 ts（$\mathrm{o} \prec^{\text{hb}}_{\mathrm{h}} ts$）之前，或者反过来 $ts \prec^{\text{hb}}_{\mathrm{h}} \mathrm{o}$。（这边所说的 A 发生在 B 之前，不仅仅代表 A 的逻辑时间要小于 B 的逻辑时间，而且还包含了两者必须满足 happens-before 关系的意思。下文中如果没有特殊说明，均用对发生加上双引号来同一个意思）。

有了这些定义，我们就能够定义出对于普通变量的读操作是如何实现的：带有时间戳 *ts* 的读操作可以读到在历史记录 h 中任何满足 “可视 (visible)” 条件的写操作 *wv*, 换句话说，即可以读到任何满足 $visible(ts, wv, \mathrm{h})$ 的 *wv*。如图 3.5中定义的那样，*wv* 要么是 “发生” 在 *ts* 之前同时又是最近的写操作，要么 *wv* 和 *ts* 之间没有任何 happens-before 关系。如果存储在历史记录中的写动作 $\langle ts, \mathrm{n}, \mu \rangle$ 被其他线程所看到（即其他线程的 ID 不等于 *ts.tid*）, 那么我们就会标记 μ 域为 **true**。因此这个写动作就会变成 $\langle ts, \mathrm{n}, \textbf{true} \rangle$。

现在我们可以知道如何通过基于历史记录的内存来允许形如例3.1.2的结果发生了。我们可以先执行语句 1 和语句 3 所对应的事件。然后，对于语句 2 所对应的读事件来说，语句 1 的写操作是 “发生” 在它之前的最近的写操作，所以语句 2 能够看到语句 1 的值，同时语句 3 的写操作和语句 2 之间没有任何的 Happens-before 关系，所以语句 3 的值也能被语句 2 看到。同理，语句 4 也能同时看到两个写操作的值。

例 3.1.3. 下面是一个简单的串行程序，我们可以先执行写操作所对应的事件，但是并不会影响最终的结果 $r_1 = 0$。

$$r_1 := \mathrm{x};\ \mathrm{x} := 1;$$

这是由于在这个例子中因为读事件先被线程发出放入事件缓存中，所以读事件的时间戳比写事件要小。因此即使我们先执行了写事件，它对 x 的历史记录写入的值 1 也不会被读事件看到。因为根据定义，它对读事件是不可见的（参见图 *3.5*中 *visible* 的定义)。事实上在这里例子中，只有初始值 0 能被读事件看到，因为初始值所对应的时间戳是 **init**, 是一个特殊的时间戳，小于任何其他由线程发出的事件的时间戳。

3.1.5　重放事件

许多编译器优化都基于程序分析的结果。而这种结果，经常会产生一些使得我们不能仅仅依靠上面提到的事件缓存和历史记录来模拟的弱行为。

例 3.1.4 (Adapted from [12].)**.**

$$\begin{array}{l|l} 1:\ r_1 := \mathrm{x}; & \\ 2:\ r_2 := r_1; & \\ 3:\ r_3 := (r_1 == r_2); & 6:\ r_4 := \mathrm{y}; \\ 4:\ \textbf{if}\ (r_3) & 7:\ \mathrm{x} := r_4; \\ 5:\quad \mathrm{y} := 42; & \end{array}$$

$$\textit{Result: } r_1 = r_2 = r_4 = 42?$$

这样的程序行为初看起来会很奇怪，但是是事实上我们应当允许这样的行为发生。在现实世界中，编译器往往会发现，第 *4* 行的条件测试永远为真，所以第 *5* 行一定会被执行。因此，第 *5* 行就会被提取出条件分支，然后因为它和第 *1*，*2* 行的语句之间没有数据依赖性，所以第 *5* 行在某些优化中会被放到第 *1*，*2* 行之前执行。这样，就会得到如上的结果。那么，在我们的模型中，如果仅仅依靠事件缓存和历史记录，我们会发现，我们必须先执行第 *1*，*2*，*3* 行，然后才能执行第 *5* 行，因为这些语句之间有寄存器依赖性（见小节 *3.1.3*)。所以只用事件缓存和历史记录，我们并不能弱化约束到能够允许这种行为。

为了让我们的模型能够模拟这样的由编译器作出的程序变换，我们可以注意到，事实上编译器在做程序分析的时候，它需要先扫描前三行，然后才能决定第 4 行的条件测试恒为真，接下来它才能做程序变换把第 1，2 行和第 5 行调序执行。而我们在抽象机中，我们可以通过复制第 1 行和第 2 行的语句然后把副本放到第 5 行之后来模拟这种变换。

$$\begin{array}{ll} 1: & r_1 := \mathrm{x}; \\ 2: & r_2 := r_1; \\ 3: & r_3 := (r_1 == r_2); \end{array} \qquad \begin{array}{ll} 4: & \textbf{if}\ (r_3) \\ 5: & \quad \mathrm{y} := 42; \\ 1': & r_1 := \mathrm{x}; \\ 2': & r_2 := r_1; \end{array}$$

我们可以发现，在这种情况下，尽管我们重复执行了第 1，2 行，但是线程的顺序行为并没有发生改变。直观上，我们在第一次执行第 1，2 行的时候，我们是在模拟编译器的静态分析过程，然后第二次执行副本（行 $1'$ 和行 $2'$）的时候则是对应于现实世界中编译器优化以后把第 5 行和第 1，2 行的调序执行。

基于上述这种观察，我们允许通过当执行一个事件的同时把它的副本放入*线程局部的重放缓存*来实现重复执行一个事件若干遍。然后接下来某个时刻，我们会把重放缓存中的事件副本从重放缓存放回事件缓存，因此这个事件又能够再次被执行了。值得注意的是，事件副本和事件本身是一模一样的，连时间戳也是完全相同的。再次回到例3.1.3, 我们注意到，即使我们复制读事件，然后在写事件之后把读事件的副本从重放缓存取回，读事件读到的值还是只能为 0，原因和我们之前在例3.1.3解释的一样，因为时间戳没有改变，所以读事件的副本还是只能读到初始值。然而，重放机制是有所限制的，如果我们不加约束的滥用，那么，就会导致串行程序的行为的改变，从而破坏我们一直想要维持的 DRF-Guarantee 性质或者产生凭空出现的值。

例 3.1.5.

$$\begin{array}{ll} 1: r_1 := 1; & 1: r_1 := 1; \\ 2: r_1 := 2; & 2: r_2 := r_1; \\ & 3: r_1 := 2; \\ \textit{(a) } r_1 = 1? & \textit{(b) } r_2 = 2? \end{array}$$

$$\begin{array}{ll} 1: r_1 := \mathrm{x}; & 4: r_2 := \mathrm{x}; \\ 2: r_1 := r_1 + 1; & 5: r_2 := r_2 + 1; \\ 3: \mathrm{x} := r_1; & 6: \mathrm{x} := r_2; \\ & 7: r_3 := \mathrm{x}; \end{array}$$

(c) $r_3 = 3$*?*

在例3.1.5(a) 中，我们可以通过重复执行第 1 行所对应的事件来得到所示的结果。在 (b) 中，我们可以通过重复执行第 2 行，但是不重复执行第 1 行，来得到结果。在 (c) 中，我们可以在执行完第 1 到第 6 行以后再重复执行第 1，2，3

行来得到结果。如此我们得到的结果 $r_3 = 3$ 是凭空出现的值，因此是不应该被一个好的模型所允许的。

为了处理这样的问题，我们应该对重放机制加上两条重要的原则约束。第一，重放不能改变串行程序的行为（用来禁止例3.1.5中形如 (a) 和 (b) 的行为出现）。第二，一个写事件能被重放的前提是它所对应的在历史记录中的写动作并没有被其他线程读取。这样，就能禁止掉例3.1.5(c) 中的行为。技术实现上，我们需要以下这些规则来实现上面所说的两条原则约束：

- 如果事件 e 读写的寄存器是被已经放在重放单元事件 e' 更新，那么 e 也必须被重放。形式化描述即: $(UseR(e.\iota) \cup UpdR(e.\iota)) \cap (UpdR(rb)) \neq \emptyset$, 这里 $UpdR(rb)$ 是由放在重访缓存中的所有事件所能够更新的寄存器的集合 (它的定义见图 3.4)。在例3.1.5(a) 中，如果由第 1 行语句所产生的事件放在重放缓存中，那么当我们执行第二行所对应的事件的时候的，我们必须也把第二行所对应的事件放到重访缓存中。因此就不可能得到 $r_1 = 1$ 这样的违反串行程序的行为。

- 如果事件 e 使用了寄存器 r(形式化即 $r \in UseR(e.\iota)$) 的值, 但之前对该寄存器进行赋值的事件并没有存放在重放缓存中 (即 $r \notin UpdR(rb)$), 那么我们一定不能重放事件 e。否则，事件 e 的副本将会看到那些由其后续指令所更新的 r 的值，从而违反了串行程序的行为。在例3.1.5(b) 中，如果我们复制第 2 行而没有复制第 1 行，那么第二行的副本将会看见 r_1 被第三行更新的值。这条规则就保证了这种情况的发生。

- 如果上面两条规则所描述的条件都不成立，我们就可以执行 e 并且非确定性的决定是否把 e 放入重放缓存 rb 中。

- 如果第 1，2 条规则所描述的条件同时成立，那么事件 e 的执行就会被阻塞，直到这两条规则所描述的条件有一条不成立。以上这四条规则的形式化描述由图 3.7中的 $Replay(rb, e, rb')$ 所定义。

- 如果一个写内存事件的副本被放进了 rb, 那么该副本最后是否能被从 rb 中取出放回事件缓存进行第二次执行，取决于它之前的执行所写入历史记录的写动作是否被其他线程所读取。即写动作的标示位 μ 是否是 **false**. 在例3.1.5(c) 中，我们可以执行第 3 行（往历史记录里面写入 1），然后把第三行的副本事件放入 rb 中，接下来执行第 4 行 (该行的事件会读到值 1 的写动作)。这时，第三行的副本就不能再被执行，因为它之前的执行所写入的值已经被其他线程所看到（因此写动作的 μ 变为 **true**。这条规则确保了形如例3.1.5(c) 中凭空出现的 $r_3 = 3$ 这样的结果不会出现。这条规则同时还解释了我们为什么会需要要求在历史记录中的写动作有这样的标示位 μ。如果写事件的副本能够顺利执行（即它之前执行所对应的写动作并没

$$\frac{C = \iota \quad \iota \neq \textbf{lock}\_ \quad e = \langle\langle \mathsf{i}, \mathsf{t}\rangle, \iota\rangle \quad \mathsf{b}' = \mathsf{b} \cup \{e\}}{(\mathbb{P}[\mathsf{i}.C], (tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L})) \longmapsto (\mathbb{P}[\mathsf{i}.\textbf{skip}], tq, m, \mathsf{b}', \mathsf{t}{+}1, \mathsf{L})} \text{ (ISSUE)}$$

$$\frac{\begin{array}{c} C = \textbf{lock}\ \mathsf{l} \quad \mathsf{l} \notin \mathsf{dom}(\mathsf{L}) \quad \mathsf{L}' = \mathsf{L}\{\mathsf{l} \rightsquigarrow \mathsf{i}\} \\ m' = \mathit{AddSyn}(m, \langle\langle \mathsf{i}, \mathsf{t}\rangle, \textbf{acq}, \mathsf{l}\rangle) \end{array}}{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.\textbf{skip}], \langle tq, m', \mathsf{b}, \mathsf{t}{+}1, \mathsf{L}'\rangle)} \text{ (LK)}$$

$$\frac{\begin{array}{c} C = (\textbf{if}\ r\ \textbf{then}\ C_1\ \textbf{else}\ C_2) \\ \mathit{readyR}(\langle \mathsf{i}, \mathsf{t}\rangle, r, \mathsf{b}) \quad tq(\mathsf{i}).\mathit{rf}(r) \neq 0 \end{array}}{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.C_1], \langle tq, m, \mathsf{b}, \mathsf{t}{+}1, \mathsf{L}\rangle)} \text{ (IF-T)}$$

$$\frac{\begin{array}{c} C = (\textbf{if}\ r\ \textbf{then}\ C_1\ \textbf{else}\ C_2) \\ \mathit{readyR}(\langle \mathsf{i}, \mathsf{t}\rangle, r, \mathsf{b}) \quad tq(\mathsf{i}).\mathit{rf}(r) = 0 \end{array}}{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.C_2], \langle tq, m, \mathsf{b}, \mathsf{t}{+}1, \mathsf{L}\rangle)} \text{ (IF-F)}$$

$$\frac{\begin{array}{c} C = (\textbf{while}\ r\ \textbf{do}\ C_1) \quad C'' = C_1; C \\ \mathit{readyR}(\langle \mathsf{i}, \mathsf{t}\rangle, r, \mathsf{b}) \quad tq(\mathsf{i}).\mathit{rf}(r) \neq 0 \end{array}}{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.C''], \langle tq, m, \mathsf{b}, \mathsf{t}{+}1, \mathsf{L}\rangle)} \text{ (WHILE-T)}$$

$$\frac{\begin{array}{c} C = (\textbf{while}\ r\ \textbf{do}\ C_1) \quad \mathit{readyR}(\langle \mathsf{i}, \mathsf{t}\rangle, r, \mathsf{b}) \\ tq(\mathsf{i}).\mathit{rf}(r) = 0 \end{array}}{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.\textbf{skip}], \langle tq, m, \mathsf{b}, \mathsf{t}{+}1, \mathsf{L}\rangle)} \text{ (WHILE-F)}$$

$$\frac{(\mathbb{P}[\mathsf{i}.C], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.C'], \langle tq', m', \mathsf{b}', \mathsf{t}', \mathsf{L}'\rangle)}{(\mathbb{P}[\mathsf{i}.C; C_1], \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (\mathbb{P}[\mathsf{i}.C'; C_1], \langle tq', m', \mathsf{b}', \mathsf{t}', \mathsf{L}'\rangle)} \text{ (SEQ)}$$

$$\frac{\begin{array}{c} tq(\mathsf{i}) = (rf, rb) \quad tq' = tq\{\mathsf{i} \rightsquigarrow (rf', rb')\} \\ \langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow{\mathsf{i}} \langle (rf', rb'), m', \mathsf{b}', \mathsf{L}'\rangle \end{array}}{(P, \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (P, \langle tq', m', \mathsf{b}', \mathsf{t}, \mathsf{L}'\rangle)} \text{ (EVT)}$$

$$\frac{tq(\mathsf{i}) = (rf, rb) \quad tq' = tq\{\mathsf{i} \rightsquigarrow (rf, \emptyset)\}}{(P, \langle tq, m, \mathsf{b}, \mathsf{t}, \mathsf{L}\rangle) \longmapsto (P, \langle tq', m, \mathsf{b} \cup rb, \mathsf{t}, \mathsf{L}\rangle)} \text{ (REPLAY)}$$

图 3.6: OHMM 操作语义：从指令到事件

有被其他线程所看到），那么这个副本的执行就不是在往历史记录里面添加新的写动作，而是直接更新原有的写动作的值。

3.2 OHMM 形式化定义

在这一节中，我们将通过对之前提供的程序语言抽象给出操作语义来形式化定义 OHMM。我们首先定义，什么是一个程序的执行上下文环境 $\mathbb{P}$, 它的意思是说我们可以在在任何一步都选择任意一个线程进行执行。

$$(\mathit{ThrdCtxt}) \quad \mathbb{P} \ ::= \ [\,] \mid \mathit{tid}.C \| \mathbb{P} \mid \mathbb{P} \| \mathit{tid}.C$$

$$Enabled(\mathfrak{b},\mathfrak{e},\mathfrak{i}) \stackrel{\text{def}}{=} (\mathfrak{e}\in\mathfrak{b}) \wedge (\mathfrak{e}.ts.tid = \mathfrak{i}) \wedge \neg\exists \mathfrak{e}' \in \mathfrak{b}.\, \mathfrak{e}' \leftarrow \mathfrak{e}$$

$$\begin{aligned} Replay(rb,\mathfrak{e},rb') \stackrel{\text{def}}{=}\ & ((UseR(\mathfrak{e}.\mathfrak{i}) \cup UpdR(\mathfrak{e}.\mathfrak{i})) \cap UpdR(rb) = \emptyset) \wedge rb' = rb \\ & \vee (UseR(\mathfrak{e}.\mathfrak{i}) \subseteq UpdR(rb)) \wedge rb' = rb \uplus \{\mathfrak{e}\} \end{aligned}$$

$$ModRef(m,\mathfrak{x},ts) \stackrel{\text{def}}{=} \lambda\mathfrak{x}'.\ \begin{cases} \mathfrak{h}\cup\{\langle ts,\mathfrak{n},\textbf{true}\rangle\} & \text{if } \mathfrak{x}'=\mathfrak{x} \\ & \wedge m(\mathfrak{x})=\mathfrak{h}\uplus\{\langle ts,\mathfrak{n},\_\rangle\} \\ m(\mathfrak{x}') & \text{if } \mathfrak{x}'\neq\mathfrak{x} \\ undef & \text{otherwise} \end{cases}$$

where $\uplus$ means the union of two disjoint sets.

$$Add(m,\mathfrak{x},ts,\mathfrak{n}) \stackrel{\text{def}}{=} \lambda\mathfrak{x}'.\ \begin{cases} m(\mathfrak{x})\cup\{\langle ts,\mathfrak{n},\textbf{false}\rangle\} & \text{if } \mathfrak{x}'=\mathfrak{x} \\ & \wedge\neg\exists\mathfrak{n}',\mu. \\ & \quad \langle ts,\mathfrak{n}',\mu\rangle\in m(\mathfrak{x}) \\ \mathfrak{h}\cup\{\langle ts,\mathfrak{n},\textbf{false}\rangle\} & \text{if } \mathfrak{x}'=\mathfrak{x} \\ & \wedge m(\mathfrak{x})=\mathfrak{h} \\ & \quad \uplus\{\langle ts,\mathfrak{n}',\textbf{false}\rangle\} \\ m(\mathfrak{x}') & \text{if } \mathfrak{x}'\neq\mathfrak{x} \\ undef & \text{otherwise} \end{cases}$$

图 3.7: OHMM 操作语义：重要谓词定义

程序执行的操作语义如图 3.6所示。对于除了 **lock** 以外的其他指令，我们将指令和线程 ID 以及当前逻辑时间包装在一起，组成我们说称的事件。然后把事件放入事件缓存中。而 **lock** 指令则是直接执行，线程并不发出任何相应的事件。我们使用 $\mathfrak{f}\{\mathfrak{x}\rightsquigarrow\mathfrak{n}\}$ 代表把函数 $\mathfrak{f}$ 定义域中的元素 $\mathfrak{x}$ 所映射的值更新为 $\mathfrak{n}$. 值得注意的，**lock** 指令会被阻塞如果它所请求的锁资源已经被其他线程占有，而当 **lock** 指令成功请求到同步锁资源后，抽象机也会同时向所有非 volatile 历史记录中加入相应到同步动作。相关的形式化表达参见图 3.5中的 *AddSyn* 定义。对于 **if** 语句，它也会被阻塞直到它的条件表达式中所用的寄存器 *r* 的值已经 “准备” 完毕（请参见图 3.4中的 *readyR* 的定义）。

规则 EVT 说明在事件缓存中的事件的执行能够和线程发出新的事件并发执行。执行一个来自于线程 $\mathfrak{i}$ 的事件的语义请参见图 3.8。最后，REPLAY 规则说明在任意时间我们都能选择将某个线程局部的重放缓存清空，并把里面原本所包含的事件副本都移动到事件缓存里面以待将来某个时候再次执行它们。

在图 3.8中的所有事件执行规则里，除 NO-WT-REPLAY 规则外，其他的规则都隐含了一个前提 $Enabled(\mathfrak{b},\mathfrak{e},\mathfrak{i})$（于图 3.7中定义）。这个前提非形式化的含义是来自于线程 $\mathfrak{i}$ 的 $\mathfrak{e}$ 和事件缓存的 $\mathfrak{b}$ 里当前所保存的所有事件都没有任何依赖性。依赖性的定义读者可以回顾一下在图 3.5中关于 $\mathfrak{e}'\leftarrow\mathfrak{e}$ 的定义。

规则 RD-V,WT-V 和 UNLK 显示了同步事件是如何执行的。我们不会对同步事件进行重放，所以重放缓存 *rb* 在这些事件执行前后不会发生改变。同时，当同步事件执行的时候，我们需要往所有的非 volatile 变量的历史记录中添加同

$$\frac{e = \langle ts, r := v\rangle \quad syn = \langle ts, \mathbf{ld}, v\rangle \quad m' = AddSyn(m, syn) \quad rf' = rf\{r \rightsquigarrow m(v)\} \quad r \notin UpdR(rb)}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf', rb), m', \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(RD-V)}$$

$$\frac{e = \langle ts, v := \mathtt{r}\rangle \quad syn = \langle ts, \mathbf{st}, v\rangle \quad m' = AddSyn(m, syn) \quad m'' = m'\{v \rightsquigarrow rf(r)\} \quad r \notin UpdR(rb)}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf, rb), m'', \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(WT-V)}$$

$$\frac{e = \langle ts, \mathbf{unlock}\ \mathtt{l}\rangle \quad \mathtt{L}(\mathtt{l}) = \mathtt{i} \quad \mathtt{L}' = \mathtt{L} \setminus \{\mathtt{l}\} \quad syn = \langle ts, \mathbf{rel}, \mathtt{l}\rangle \quad m' = AddSyn(m, syn)}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf, rb), m', \mathtt{b}\backslash\{e\}, \mathtt{L}'\rangle} \text{(UNLK)}$$

$$\frac{e = \langle ts, r := \mathtt{x}\rangle \quad visible(ts, \langle ts', \mathtt{n}, \_\rangle, m(\mathtt{x})) \quad ts.tid = ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad Replay(rb, \mathtt{e}, rb')}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf', rb'), m, \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(RD-SELF)}$$

$$\frac{e = \langle ts, r := \mathtt{x}\rangle \quad visible(ts, \langle ts', \mathtt{n}, \_\rangle, m(\mathtt{x})) \quad ts.tid \neq ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad m' = ModRef(m, \mathtt{x}, ts') \quad Replay(rb, \mathtt{e}, rb')}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf', rb'), m', \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(RD-OTHER)}$$

$$\frac{e = \langle ts, \mathtt{x} := r\rangle \quad e \in \mathtt{b} \quad \langle ts, \_, \mathbf{true}\rangle \in m(\mathtt{x})}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf, rb), m, \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(NO-WT-REPLAY)}$$

$$\frac{e = \langle ts, \mathtt{x} := r\rangle \quad rf(r) = \mathtt{n} \quad m' = Add(m, \mathtt{x}, ts, \mathtt{n}) \quad Replay(rb, \mathtt{e}, rb')}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf, rb'), m', \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(WT)}$$

$$\frac{e = \langle ts, \mathtt{r} := \mathtt{E}\rangle \quad [\![E]\!]_{rf} = \mathtt{n} \quad rf' = rf\{\mathtt{r} \rightsquigarrow \mathtt{n}\} \quad Replay(rb, \mathtt{e}, rb')}{\langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow{\mathtt{i}} \langle (rf', rb'), m, \mathtt{b}\backslash\{e\}, \mathtt{L}\rangle} \text{(PURE)}$$

图 3.8: OHMM 操作语义：事件执行

步动作，以用来让读事件判断哪些写动作是可见的。我们用 *AddSyn*（参见图 3.5）来代表把这些同步事件相应的同步动作添加到内存中所有非 volatile 变量的历史记录中。

规则 RD-SELF 表达了当一个普通变量从历史记录中读到跟它来自同一个线程的写动作时的规则。写动作的 “可见性” 在图 3.5中定义。$Replay(rb, \mathtt{e}, rb')$ 这个定义在图 3.7中的谓词是用来计算是否把 e 放入重放缓存 rb 中，当需要把 e 放

入重放缓存的时候，$rb' = rb \uplus \{e\}$; 而当 e 不能放入缓存的时候，$rb' = rb$。我们在小节 3.1.5已经解释过这一点。

规则 RD-OTHER 则是用来推导当一个读事件读到来自于其他线程的写动作时的情形。在这个规则中，我们会通过这个 $ModRef(m, \chi, ts)$（同样定义于图 3.7中）谓词来把被读到的来自其他线程的写动作的标示位 μ 置为 **true**，这样等我们下次再检视这个写动作的时候，我们就知道它已经被其他线程所读取了。

如果我们想执行一个写事件并且注意到这时在相应的内存单元的历史记录中已经存在一个时间戳和这个写事件一样的写动作，那么我们就知道，这个写事件一定是一个之前写事件的重放。规则 NO-WT-REPLAY 说的是如果其对应的在历史记录中的写动作已经被其他线程所读取的话，我们必须直接丢弃这个写事件 (即从事件缓存中删除这个事件并且不执行它)。回顾一下我们在小节 3.1.5中的解释，这个丢弃规则相当重要，因为它可以避免我们的模型出现凭空出现的值。如果并没有这么一个写动作在历史记录中，我们可以通过规则 WT 来执行这个写事件。而是否同时把这个写事件放入重放缓存 rb 中同样取决于谓词 $Replay(rb, e, rb')$。最后一条规则 PURE 则是用来处理 “纯指令”（形如 “$r := E$”, 回顾小节 3.1.1中的解释）产生的事件。

3.3　一些简单的例子

在这一节中，我们将会展示更多的例子用来帮助读者更好理解我们的模型。同时，我们还会在某几个例子上讨论程序在我们模型与 JMM 上所产生的不同的程序行为，以此来说明 OHMM 在这些例子比 JMM 上更加符合程序员的直观认识。如节 3.1中假设的那样，我们在这边仍然假设，下面所有例子的抽象机在初始化状态的时候，所有的内存单元都为 0(对于普通变量来说，则是历史记录中有且只有一条时间戳为 **init** 且写入值为 0 的写动作)。

回顾例2.2.2。 首先回顾下我们在第二章中提到的，HMM 的因果循环会导致例子中这样奇怪的结果的发生。而这样的行为在我们的模型上不被允许的。首先，对于这个例子左边的程序，在我们的模型上，根据我们上一节定义的操作语义，我们知道第 2 行和第 3 行不能在第 1 行之前执行 (寄存器依赖性，见小节 3.1.3)，所以第 1 行将会读到 0，并且使得第 2 行的条件测试为假。因此第 3 行将不会被执行。同理，第 4 行的语句所对应的事件也只能读到 0。事实上这个程序是无数据竞争的程序，因此一个完备的弱内存模型应当保证这样的程序在其之上所能出现的结果应当和在顺序一致性模型上一致。我们将会在节 3.4中证明我们的模型是满足 DRF-Guarantee 的，所以我们的模型能够保证这一点, $r_1 = r_2 = 0$ 是这个程序在我们模型上唯一可能出现的结果。而对于右边的程序来说，因为 OHMM 是一个按照操作语义自然生成的模型，因此 r_1 和 r_2 除了 0 之外不可能有其他的值，因此我们也不会出现像 HMM 那样凭空出现的值。

接下来一个例子，我们想再用来强调这么一点：我们的抽象机仅仅是设计用来模拟程序行为的。我们并不是想将这些构件实际用于现实世界的硬件或者软件优化。因此，我们语义中的调换事件顺序，或是重放机制等，并不是和现实世界中的优化是一一对应的，但是我们通过这些语义，我们能够让程序在我们模型上的行为模拟现实世界中优化后的行为。

例 3.3.1.

$$\begin{array}{l} 1 : r_1 := x; \\ 2 : r_2 := 10 + 0 * r_1; \\ 3 : y := r_2; \end{array} \quad \Big\| \quad \begin{array}{l} 4 : r_3 := y; \\ 5 : r_4 := 10 + 0 * r_3; \\ 6 : x := r_4; \end{array}$$

Result: $r_1 = r_3 = 10$?

上面这个例子中，在现实世界的编译器的分析中，编译器会分析出第 *2* 行 $0 * r_1$ 恒为 *0,* 因此 r_2 的值事实上是一个常量 *10*，同理，它也会判断出 *r4* 是常量 *10*。于是编译器可能会做下面的变换:

$$\begin{array}{l} r_1 := x; \\ y := 10; \end{array} \quad \Big\| \quad \begin{array}{l} r_3 := y; \\ x := 10; \end{array}$$

因此，左右两个线程分别的两个语句都变成了没有数据依赖性的语句，编译器随时可能对他们进行调序，然后得到上述的 $r_1 = r_3 = 10$ 的结果。读者读到这边，可能会这么想: *“因为我们的抽象机里面，没有对这一层面的语法进行分析,* 即我们没法判断出，第 *2* 行，第 *5* 行是常量赋值，因此，对于我们的语义来说，第 *3* 行总是和第 *1* 行有数据依赖性的，同理第 *6* 行总是依赖于第 *4* 行的，我们无法将其调序而得到上述的结果*”*。是的，我们的确无法将其进行调序，但是我们仍然可以通过重放机制来模拟这种优化行为。在我们的抽象机中，尽管我们无法对第 *1*，*2*，*3* 行以及第 *4*，*5*，*6* 行之间进行调序，但是我们可以先按序执行第 *1*，*2*，*3* 行，让 r_1 读到 *0,* 并把第 *1*，*2*，*3* 行放入重放缓存中，然后执行第 *4*，*5*，*6* 行，让 r_4 读到 *10*（来自于第 *3* 行的写入)。最后，我们重新执行第 *1* 行，这一次我们让第 *1* 行读到 *10* 来得到我们想要模拟的结果。

接下来的例子中，根据 OHMM 与 JMM 的不同之处，我们将这些例子大致分为两类:

3.3.1　JMM 中不允许但 OHMM 中允许

例 3.3.2 (Causality Test Case 5 [30])**.**

$$\begin{array}{l} 1 : r_1 := x; \\ 2 : y := r_1; \end{array} \quad \Big\| \quad \begin{array}{l} 3 : r_2 := y; \\ 4 : x := r_2; \end{array} \quad \Big\| \quad 5 : z := 1; \quad \Big\| \quad \begin{array}{l} 6 : r_3 := z; \\ 7 : x := r_3; \end{array}$$

Result: $r_1 = r_2 = 1, r_3 = 0$?

上面这个例子，来自于 *JMM* 中的提到的因果测试例 *[30]*。因果测试例中列举了 *20* 来个例子，并且探讨其行为是否应当被允许。对于这里面的大部分例子，我们的模型和 *JMM* 表现出来的一样。然而，有几个例子是否应当禁止或者允许，还是存有异议的，比如 *JMM* 认为如上的程序结果应该被认作是凭空出现的值，因此这样的程序行为被其所禁止。然而，这个例子我们认为它并不像例*2.2.2*中的行为那样*“不好”*，因为最后寄存器读到的值 1 确实出现在程序里面并被写入

变量 z 中。这个程序行为是否应该被允许或者禁止在 *JMM* 相关的邮件讨论列表中是非常具有争论性的。在我们的模型上，这样的程序行为是被允许的。我们假定第 *1* 行到第 *7* 行所产生对应的事件是 $e_1, e_2, \ldots, e_7$。并且我们把这些事件都一一放入事件缓存中而暂不执行它们。然后我们首先执行 e_5，接下来执行 e_6 然后同时把它放入重放缓存中。我们让 r_3 从 z 中读到 *1*（根据我们在图 *3.5*中对*"可见"*的定义，事实上初始值 *0* 和事件 e_5 写入的 *1* 都是对 e_6 可见的，至于它读哪一个则是不确定的。我们在这边人为的选择让它选择读到 *1*）。下一步我们按照 e_7, e_1, e_2, e_3, e_4 的顺序来执行剩下的事件，然后让 r_1 和 r_2 都读到 *1*（同理，我们这里也是人为的让它们选择读到 *1*）。最后，我们把 e_6 的副本从重放缓存中取出并且再次执行它，这一次，我们让它选择读到 *0*, 即 z 的初始值 *0*，就像我们前面讲过的那样，*0* 和 *1* 这两个值对 e_6 都是可见的，读哪一个是不确定的，不过我们这一次人为的让它选择读到 *0* 来让程序产生上述的结果。因果测试例 *Causality Test Case 10 [30]* 是一个和这个例子非常类似并且同样是相当有争议性的被 *JMM* 所禁止的程序。同样，这个例子在我们模型下也是允许的。

下一个例子来自于因果测试例 Causality Test Case 17 [30]，JMM 宣称它允许这种例子的结果，但是被指出其中有一个非常小的 bug 导致其并不允许产生这样的结果 [26]。

例 3.3.3 (Causality Test Case 17 [30])**.**

$$\left(\begin{array}{ll} 1: r_1 := x; & \\ 2: r_2 := (r_1 \texttt{!=} 42); & 5: r_3 := x; \\ 3: \textbf{if}(r_2) & 6: y := r_3; \\ 4: \quad x := 42; & \end{array}\right) \Bigg\| \begin{array}{l} 7: r_4 := y; \\ 8: x := r_4; \end{array}$$

Result: $r_1 = r_3 = r_4 = 42$*?*

这个例子的行为在我们的模型下是允许的。我们可以先执行语句 *1*（所对应的事件），读到 *0*，然后同时将其放入重放缓存中。接下来我们进入到 *if* 语句的分支并且向 x 的历史记录中添加值为 *42* 的写动作。然后我们接着执行其他语句，得到 $r_3 = r_4 = 42$, 并且在执行语句 *8* 所对应的事件时时对 x 的历史记录添加另一个 *42* 的写动作。最后，我们再次执行语句 *1* 事件的副本，这时根据我们的语义，它可以从历史记录中读到由语句 *8* 事件写入的 *42*。（注意，这时历史记录中有两个写动作都写入了 *42*，由语句 *4* 写入的 *42* 并不能被语句 *1* 事件的副本读到，因为语句 *1*"发生"在它之前）。*JMM* 宣称它也支持这样的行为产生，但是事实上 *Aspinall* 在 *[26]* 中指出 *JMM* 并不能支持这样的行为，因为它的模型定义里面有一处细微的 *bug*（同样也是因为 *JMM* 的复杂导致这个 *bug* 之前一直没有被发现）。并且由于同样的 *bug*，*JMM* 同样不能向它宣称的那样支持测试例 *18-20 [30]* 的行为, 而这些例子 *OHMM* 均允许这样的行为。

例 3.3.4 ("bait-and-switch" behaviors)**.**

$$\begin{array}{l} 1: r_1 := x; \\ 2: r_2 := (r_1 \texttt{==} 0); \\ 3: \textbf{if}(r_2) \\ 4: \quad x := 1; \end{array} \Bigg\| \begin{array}{l} 5: r_3 := x; \\ 6: y := r_3; \end{array} \Bigg\| \begin{array}{l} 7: r_4 := y; \\ 8: x := r_4; \end{array}$$

Result: $r_1 = r_3 = r_4 = 1$*?*

这个例子展示了 *bait-and-switch behavior(Fig.11 in [12])*，如上的结果在 *JMM* 中是被禁止的。然而，如果我们把左边和中间这两段代码的结合到一起放在同一个线程中，即把中间的代码放到左边代码的后面*:*

```
1 : r1 := x;                  ||
2 : r2 := (r1 == 0);          ||
3 : if(r2)                    ||
4 :     x := 1;               ||   7 : r4 := y;
5 : r3 := x;                  ||   8 : x := r4;
6 : y := r3;                  ||
```

那么显然，我们得到的新的程序应该比其原本的程序拥有更少的并发性*(*因为线程数量减少了，从三个变成两个*)*，但是，此时新的程序反而允许上述行为的发生。这是一个令人惊讶并且违反直观的性质，即由同样的语句组成，拥有更少并发性的程序反而比拥有更多并发交互可能的程序所可能产生的行为要多。

反之，在我们的模型当中，不管是原本的程序还是新的程序，都是允许如上提到的结果的，这样更符合我们的直观认识。对于原本的程序来说，我们可以先执行左边代码中的语句，并且同时把语句 *1* 事件副本放入重放缓存中。然后我们接着按序执行中间和右边的代码。最后，我们再次执行语句 *1*，此时我们就能让它从历史记录中读到 *1*（这个写动作来自于第 *8* 行）。

事实上允许原本的程序出现这样的结果，看起来是没有坏处的。如同 *Manson* 等人在 *[12]* 中指出的那样，他们在 *JMM* 中禁止这样的行为更多的是因为 *"taste and preference"* 而不是因为有确切的证据说明这样的程序行为不好。事实上，还有另外一个类似的例子被 *JMM* 所禁止但是被我们的模型所允许（*Fig. 10 in [12]*）。

3.3.2　OHMM 中禁止 JMM 中违反直观的例子出现

JMM 不仅有上述如几个宣称能够允许但实际不能允许例子，还有另外一个问题，就是有些它所允许的程序行为明显违背了人们的直观认识。下面的例子来自于 Cenciarelli 的文章 [13]，同时也被 Aspinall and Ševčík [14] 用来举例 "the ugly part of JMM"。

例 3.3.5.

```
1 : r1 := z;                  ||
2 : r2 := (r1 == 1);          ||
3 : if(r2){                   ||   9 :  r3 := x;
4 :     x := 1;               ||   10 : r4 := y;
5 :     y := 1;}              ||   11 : r5 := (r3==1&&r4==1);
6 : else{                     ||   12 : if(r5)
7 :     y := 1;               ||   13 :     z := 1;
8 :     x := 1;}              ||
```

Result: $r_1 = r_3 = r_4 = 1$*?*

JMM 不允许如上的程序行为发生，但是如果我们把语句 *4* 和语句 *5* 交换位置（或是语句 *7* 和语句 *8*），那么如上的程序行为就可能发生。这个例子说明在 *JMM* 中，如果我们调序两个互相独立的指令，可能会产生更多的行为。

而我们的模型不论我们是否调换 *IF* 语句的某个分支的语句顺序，都能产生相同的结果：我们可以先执行第 *1* 行，读到 *0*，然后把它放入事件缓存中。然后我们按照顺序执行来执行剩下的语句，让 r_3 和 r_4 读到 *1*. 最后，我们再次执行第 *1* 行的副本，这次我们让它读到 *1*（写动作来自第 *13* 行）。

下一个例子也是来自于 [14]。如下所示的结果是被 JMM 所禁止的。然而如果我们把第 7 行移动到它后面的临界区中，或者是把普通变量 x 换成 volatile 变量，这样的程序行为在 JMM 就是被允许的。这是另外一个 JMM 违反直观的例子: 对一个程序增加更多的同步语句，JMM 会产生更多的行为而不是减少或者死锁。

例 3.3.6 (Roach Motel Semantics)**.**

1 : **lock** l; 2 : x := 2; 3 : **unlock** l;	4 : **lock** l; 5 : x := 1; 6 : **unlock** l;	7 : r_1 := x; 8 : **lock** l; 9 : r_2 := z; 10 : r_3 := (r_1 == 2); 11 : **if**(r_3) 12 : y := 1; 13 : **else** 14 : y := r_2; 15 : **unlock** l;	16 : r_4 := y; 17 : z := r_4;

Result: $r_1 = r_2 = r_4 = 1$*?*

这个程序结果在我们的模型中是被允许的。我们可以先执行第 *1-3* 行，然后执行第 *7* 行并且把它的副本放入重放缓存。我们让第 *7* 行第一次执行时读到 *2*，然后执行第 *4* 到 *6* 行，第 *8* 到第 *12* 行（由于 r_1 等于 *2*，我们可以进入 *IF* 语句的第一个分支中），然后我们执行第九行并且把它的副本也放入重放缓存中。接下来我们执行第 *16* 行和第 *17* 行，得到 $r_4 = 1$. 在我们执行第 *15* 行之前，我们可以把第 *7* 行和第 *9* 行的副本从重放缓存中取出并且第二次执行他们。这一次我们让第 *7* 行读到 *1*（该写动作来自于第 *5* 行），让第 *9* 行也读到 *1*（该写动作来自第 *18* 行）。因此，我们可以得到 $r_1 = r_2 = r_4 = 1$。

然而，如果我们把第 *7* 行放入临界区中，我们就不能再次得到这样的结果了。因为第 *7*，*2*，*5* 行现在有一个全局的 *happens-before* 关系。重放第 *7* 行不会让它读到不同的值。（即它读不到第 *5* 行的写入了，因为第 *5* 行“发生”于它之前，该写动作对它来说是不可见的）。

同理，我们让普通变量 x 变成 *volatile* 变量也是一样，会让这样的行为不能在我们的模型下出现，因为我们现在不能重放第 *7* 行了（它现在变成一个同步事件，不能重放）。显然，在这个例子上，我们的模型表现出来的行为比 *JMM* 更加**合理**，即我们的语言能够保证，对程序增加了同步操作，会导致程序可能发生的行为减少，甚至死锁。

例 3.3.7.

1 : r_1 := y; 2 : x := r_1;	3 : **lock** l; 4 : r_2 := x; 5 : z := 1; 6 : **unlock** l;	7 : **lock** l; 8 : y := 1; 9 : r_3 := z; 10 : **unlock** l;

$$\textit{Result: } r_1 = r_2 = r_3 = 1?$$

这是另外一个来自于 *[14]* 的令人对 *JMM* 产生的令人匪夷所思的例子。因为产生这样的结果看起来就像允许被同一个锁所保护的两个临界区能够交叉执行一样 *(*显然正常情况下，被同一个锁锁保护的不同临界区，应当是执行完其中一个中的所有语句，释放锁资源以后，另一个临界区才能申请到锁资源然后执行*)*。我们的模型则不会产生这样的行为，因为根据 $\overset{b}{\leftarrow}$ 和 $\overset{s}{\leftarrow}$ *(*参见图 *3.4)* 所定义的依赖性，最后两个线程的交错执行是不能被允许的。

小结: 与 JMM 的对比。 就像我们之前列的例子所显示的那样，我们的模型的约束并不比 JMM 的要弱，反过来，JMM 模型的约束也不比我们的要弱。对于那些没有同步锁或者 volatile 变量的程序来说，我们的模型比其 JMM 要弱，并且允许许多 JMM 所不允许的行为。我们想强调的是，我们的模型可以支持所有的因果测试例 [30], 除了 case 5 和 case 10(我们在前面的例3.3.2中解释过为什么我们允许 case 5 和 case 10 所产生的行为而 JMM 不允许)。同样，我们还允许 “bait-and-switch” 这个被 JMM 所禁止的行为（见例3.3.4）。这些我们所允许而 JMM 禁止的例子，事实上在 JMM 的邮件列表讨论中是相当有争议的，编译器和现实世界中的处理器是否应当允许或者禁止这样的行为，仍然是有争议的。我们在这边相信，允许这样的行为，是无害的。

对于那些有数据竞争同时又引入了同步锁或者 volatile 变量的程序, 有一些程序行为在 JMM 中允许而在我们的模型中是禁止的 (参见例3.3.6 和例3.3.7)。如同 Aspinall 和 Ševčík 在 [14] 中所指出的那样，这些行为都是相当违反直观并且理应禁止的。

一些关于 OHMM 的更多的介绍。 这一节到这里，我们已经把 OHMM 的特性做了详细的介绍，如果读者还有兴趣的话，可以阅读下一节中我们对 OHMM 满足 DRF-Guarantee 性质的证明。除了手工证明以外，我们同时还在 Coq 工具中完成证明。节 3.5则是我们来讨论下哪些常见的用于优化的串行编译器代码变换在我们的模型下的正确性。另外，我们还提供了一个解释器工具，可以根据我们的操作语义解释运行简单的代码，并给出相应的结果的集合。机器证明代码以及解释器工具可以在 [31] 中找到。

3.4　DRF-Guarantee 性质证明

和 JMM 一样，OHMM 也有 DRF-Guarantee 性质，即对于没有数据竞争的程序来说，在 OHMM 下的行为和顺序一致性模型下一致。在这一节中，我们将给出我们对 “无数据竞争 (Data-Race-Freedom, 下文简称为 DRF)” 的形式化定义和我们对与 DRF-Guarantee 性质的证明。更多的证明细节我们列在附录 A中。除了手工证明之外，我们还使用 Coq 证明辅助工具进行了机器证明，相关的证明代码可以在 [31] 找到。

$$
\begin{array}{lrcl}
(\mathit{ThreadQ}) & tq & ::= & \{tid_0 \rightsquigarrow rf_0, \ldots, tid_k \rightsquigarrow rf_k\} \\
(\mathit{Mem}) & m & ::= & \{x \rightsquigarrow n_0, y \rightsquigarrow n_1, z \rightsquigarrow n_3, \ldots, \\
 & & & \quad v_0 \rightsquigarrow n', v_1 \rightsquigarrow n'', \ldots\} \\
(\mathit{State}) & \Sigma & ::= & \langle tq, m, t, L\rangle
\end{array}
$$

图 3.9: 强抽象机

3.4.1　无数据竞争

无数据竞争性质需要在顺序一致性模型下定义，因此，我们首先定义一个强抽象机来按照标准的顺序语义来执行程序。这个抽象机的结构定义如图 3.9所示。我们定义了 SC 模型的机器状态 Σ, 称之为强状态。这时一个四元组，由线程队列，内存，计时器以及锁资源集合组成。线程队列在这个模型中是一个由线程 ID 映射到它的寄存器文件（即保存所有寄存器的值，参见节 3.1中的图 3.1的定义）的部分函数。强状态的内存定义是标准定义，即从变量 (volatile 变量和普通变量) 到整数值的部分函数。

图 3.10中是强抽象机的操作语义，这是一个在每步执行上都加上了标签的标准顺序语义（注意，即使没有标签，这个操作语义仍然是正确的。标签在这边的作用只是为了让我们对数据竞争定义更加简洁）。每个标签是一个三元组，由动作，线程 ID 和逻辑时间组成。有几种不同的动作类型，分别标记在执行中不同的动作: 对一个普通变量进行读写 (**W** x 和 **R** x); 同步动作，包括申请锁资源 (**acq** l), 释放锁资源 (**rel** l), 从 volatile 变量中读取 (**ld** v) 和对 volatile 变量进行写操作 (**st** v); 计算表达式并将值赋予某个寄存器 (**exp** r); 最后是在 IF 和 WHILE 语句中通过条件寄存器 r 来决定取哪一个分支往下执行 (**cmp** r)。

$$
\begin{array}{ll}
(\mathit{Action}) & a := \mathbf{W}\, x \mid \mathbf{R}\, x \mid \mathbf{acq}\, l \mid \mathbf{rel}\, l \mid \mathbf{ld}\, v \mid \mathbf{st}\, v \mid \mathbf{exp}\, r \mid \mathbf{cmp}\, r \\
(\mathit{Label}) & lab := (a, tid, t)
\end{array}
$$

我们把强抽象机的每一步转换的传递反射闭包用 $(P, \Sigma) \rightarrow^* (P', \Sigma')$ 标记。一个强执行序列 $\mathscr{S}$ 是一串连续的强抽象机转换序列:

$$
\mathscr{S} = (P^0, \Sigma^0) \xrightarrow{lab_0} (P^1, \Sigma^1) \ldots \xrightarrow{lab_{n-1}} (P^n, \Sigma^n)
$$

现在我们就能够在强执行这个序列上定义 “内存冲突”。我们称，在同一个强执行序列中，两个内存操作发生了冲突，当且仅当它们来自于不同的线程，访问同一个普通变量，并且其中这少有一个是写操作。注意，内存冲突仅仅发生在普通变量上，对于 volatile 变量来说，由于对它们的读写都是同步操作，是有一个全局顺序的，因此不会产生冲突。同理，下文中提到的数据竞争也只是发生在非 volatiel 变量上。我们用 # 来标示冲突关系，这是一个作用在强执行序列的标签上二元关系:

$$\frac{tq(\mathtt{i})(r) = \mathtt{n} \quad C = \mathtt{x} := r; C' \quad \mathtt{m}' = \mathtt{m}\{\mathtt{x} \rightsquigarrow \mathtt{n}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{W}\,\mathtt{x},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m', \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{m(\mathtt{x}) = \mathtt{n} \quad tq(\mathtt{i}) = \mathtt{rf} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\} \quad C = r := \mathtt{x}; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{R}\,\ell,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{C = r := E; C' \quad tq(\mathtt{i}) = rf \quad [\![E]\!]_{rf} = \mathtt{n} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{exp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = \mathtt{n} \quad C = \mathtt{v} := r; C' \quad \mathtt{m}' = \mathtt{m}\{\mathtt{v} \rightsquigarrow \mathtt{n}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{st}\,\mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m', \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{m(\mathtt{v}) = \mathtt{n} \quad tq(\mathtt{i}) = \mathtt{rf} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\} \quad C = r := \mathtt{v}; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{ld}\,\mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = 0 \quad C = \mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_2; C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) \neq 0 \quad C = \mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_1; C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = 0 \quad C = \mathbf{while}\ r\ \mathbf{do}\ C_1; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) \neq 0 \quad C = \mathbf{while}\ r\ \mathbf{do}\ C_1; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_1; C], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{C = \mathbf{lock}\ \mathtt{l}; C' \quad \mathtt{l} \notin \mathsf{dom}(\mathtt{L}) \quad \mathtt{L}' = \mathtt{L}\{\mathtt{l} \rightsquigarrow \mathtt{i}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{acq}\,\mathtt{l},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}'\rangle)}$$

$$\frac{C = \mathbf{unlock}\ \mathtt{l}; C' \quad \mathtt{L}(\mathtt{l}) = \mathtt{i} \quad \mathtt{L}' = \mathtt{L} \setminus \{\mathtt{l}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{rel}\,\mathtt{l},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}'\rangle)}$$

图 3.10: 带标签的交叉语义

定义 3.4.1 (Conflict).

$$(a_k, tid_k, t_k)\#(a_j, tid_j, t_j) \stackrel{def}{=} tid_k \neq tid_j \wedge (((a_i, a_j) = (\boldsymbol{W}\, x, \boldsymbol{R}\, x)) \\ \vee((a_i, a_j) = (\boldsymbol{W}\, x, \boldsymbol{W}\, x)) \vee ((a_i, a_j) = (\boldsymbol{R}\, x, \boldsymbol{W}\, x)))$$

我们把程序 P 和机器状态组成的二元称之为程序配置。如果一个强程序配置 (P, Σ) 是 DRF 的，当且仅当对于从它开始的每个执行序列中，如果两个两个标签 1 和 2 来自于同一个执行序列，并且它们产生了冲突，则在该执行序列上这两个标签之间，一定存在一组标签，能够构成一条 Release-Acquire 链 (Def. 3.4.3)。

这样定义的宏观上意思如果我们能够找到这么一条链，说明我们能够找到这两个内存访问之间有 Happens-before 关系，因此能够定义出先后关系从而不存在竞争关系。如果我们找不到，则说明这两个内存访问发生了竞争。

形式化的定义如下：

定义 3.4.2 (Release-Acquire Pair). 如果存在两个标签 $lab_k = (a_k, tid_k, t_k)$ 和 $lab_j = (a_j, tid_j, t_j)$ 在同一次执行序列中：

$$\mathscr{S} = (P^0, \Sigma^0) \xrightarrow{lab_0} \ldots (P^k, \Sigma^k) \xrightarrow{lab_k} \ldots (P^j, \Sigma^j) \xrightarrow{lab_j} \ldots (P^n, \Sigma^n)$$

使得:

$$lab_k.t < lab_j.t \wedge lab_k.tid \neq lab_j.tid \\ \wedge((lab_k.a = \textbf{rel}\ l \wedge lab_j.a = \textbf{acq}\ l) \vee (lab_k.a = \textbf{st}\ v \wedge lab_j.a = \textbf{ld}\ v))$$

, 那么我们说这两个标签组成了一个 *Release-Acquire Pair,* 用 $(lab_i \lessdot lab_j)$ 表示。

定义 3.4.3 (Release-Acquire Chain). 如果存在这么一串 *Release-Acquire pairs*:

$$lab_1 \lessdot lab_2, lab_3 \lessdot lab_4, \ldots, lab_{2n-1} \lessdot lab_{2n}$$

按序出现在同一个强执行系列中，并且我们有:

$$\forall i \in (0, n), lab_{2i}.tid = lab_{2i+1}.tid$$

则我们说从 lab_1 到 lab_{2n} 组成了一个 *Release-Acquire* 链, 用 $Rel\text{-}Acq_{lab_1}^{lab_{2n}}$ 表示。

定义 3.4.4 (Data Race). 对于任意的强执行序列:

$$(P^0, \Sigma^0) \xrightarrow{lab_0} (P^1, \Sigma^1) \ldots \xrightarrow{lab_{n-1}} (P^n, \Sigma^n)$$

如果存在 k, j 使得 $k < j$, $lab_k \# lab_j$ 并且不存在这么一个 *Release-Acquire* 链 $Rel\text{-}Acq_{lab_p}^{lab_q}$ 使得 $k < p < q < j$, $lab_p.tid = lab_k.tid$ 以及 $lab_q.tid = lab_j.tid$, 那么我们说这个执行序列发生了数据竞争。

直观上，数据竞争意味着有两个对普通变量的内存访问，它们来自于不同的线程，并且发生的冲突，并且它们之间并不存在 happens-before 关系来决定它们的先后。现在我们可以给出 DRF 的定义：

定义 3.4.5 (Data-Race-Free). 给定一个程序配置 (P, Σ), 对于从它开始的任何的执行序列 $\mathscr{S}$，如果都不存在任何数据竞争，那么 (P, Σ) 就是一个无数据竞争的程序配置，用 $RF(P, \Sigma)$ 表示。如果对于任意的 Σ, 都有 $RF(P, \Sigma)$ 成立，那么这个程序 P 即一个无数据竞争的程序。

3.4.2　带标签的弱操作语义

为了和带标签的顺序一致性交叉语义作比较，我们同时也需要给我们的 OHMM 的操作语义加上标签。下文为了表达方便，把 OHMM 的操作语义称之为弱操作语义，把 OHMM 的执行序列称之为弱执行序列，把 OHMM 的机器状态称之为弱机器状态。

为了给图 3.8中执行事件缓存中的事件的弱语义加上标签，上一节的标签的动作类型不够用了（因为强语义中没有事件缓存的概念），我们添加了一种新的动作，称之为缓存动作，用 *buffer action* 表示, 并扩展了 *Action* 的定义：

$$
\begin{array}{lrcl}
(\textit{Action}) & \mathfrak{a} & := & \mathbf{W}\,\mathfrak{x} \mid \mathbf{R}\,\mathfrak{x} \mid \mathbf{acq}\,\mathfrak{l} \mid \mathbf{rel}\,\mathfrak{l} \\
 & & & \mid \mathbf{ld}\,\mathfrak{v} \mid \mathbf{st}\,\mathfrak{v} \mid \mathbf{exp}\,\mathfrak{r} \mid \mathbf{cmp}\,\mathfrak{r} \mid \mathbf{BufA}\ \mathit{ba} \\
(\textit{Buffer Action}) & \mathit{ba} & := & \mathbf{Emp}\,\mathfrak{i} \mid \mathbf{Exe}\ e \mid \mathbf{Rep}\ e \mid \mathbf{Del}\ e
\end{array}
$$

同样，缓存动作也被分为几种不同的类型：执行一个事件 e(**Exe** e), 重放一个事件 e(**Rep** e), 清空线程 $\mathfrak{i}$ 的重放缓存 (**Emp** $\mathfrak{i}$) 以及从事件缓存中丢弃无用的写事件（$\mathtt{Del}\ e$, 参见图 3.8中的规则）。带标签的操作语义并没有列在正文中（因为和节 3.2中的没有区别，我们只是加上了标签以用来接下来的证明方便），而是放在了附录 A中的 图 A.2 和 图 A.3。

3.4.3　模拟关系 (Simulation)

在这一个小节中我们要把强机器状态 Σ 和弱机器状态 σ 联系起来。我们定义函数 $\mathit{Value}(\sigma,\mathfrak{x})$, 用来获得弱状态内存上某个变量 $\mathfrak{x}$ 的值最近的写入值。

$$
\mathit{Value}(\sigma,\mathfrak{x}) \stackrel{\text{def}}{=} \begin{cases}
\mathfrak{n} & \text{if } \sigma.m(\mathfrak{x}) = \mathfrak{n} \\
wv.\mathfrak{n} & \text{if } \sigma.m(\mathfrak{x}) = \mathfrak{h} \wedge wv \in \mathfrak{h}, \\
 & \quad \wedge(\forall wv', wv'' \in \mathfrak{h}. wv' \prec^{\mathrm{hb}}_{\mathfrak{h}} wv'' \vee wv' \prec^{\mathrm{hb}}_{\mathfrak{h}} wv'') \\
 & \quad \wedge(\forall wv' \in \mathfrak{h}.\ wv' = wv \vee wv' \prec^{\mathrm{hb}}_{\mathfrak{h}} wv)
\end{cases}
$$

值得注意的是，对于非 volatile 变量 $\mathfrak{x}$, $\mathit{Value}(\sigma,\mathfrak{x})$ 只有当 $\mathfrak{x}$ 的历史记录 $\sigma.m(\mathfrak{x})$ 中有基于 Happens-before 关系的全序关系作用在所有的写动作上。非常容易证明，对于 DRF 程序，在程序的弱执行过程中这样的条件总是满足的。

接下来是强机器状态和弱机器状态上的等价关系:

$$
\begin{aligned}
\sigma \stackrel{\mathsf{T}}{=} \Sigma &\stackrel{\text{def}}{=} \sigma.\mathfrak{t} = \Sigma.\mathfrak{t} \\
\sigma \stackrel{\mathsf{R}}{=} \Sigma &\stackrel{\text{def}}{=} \forall \mathfrak{i}.\ \sigma.\mathit{tq}(\mathfrak{i}).\mathit{rf} = \Sigma.\mathit{tq}(\mathfrak{i}) \\
\sigma \stackrel{\mathsf{M}}{=} \Sigma &\stackrel{\text{def}}{=} \forall \mathfrak{x}.\ \mathit{Value}(\sigma,\mathfrak{x}) = \Sigma.m(\mathfrak{x}) \\
\sigma \stackrel{\mathsf{L}}{=} \Sigma &\stackrel{\text{def}}{=} \sigma.\mathsf{L} = \Sigma.\mathsf{L} \\
\sigma \stackrel{\mathit{MRLT}}{=\!=\!=} \Sigma &\stackrel{\text{def}}{=} \sigma \stackrel{\mathsf{M}}{=} \Sigma \wedge \sigma \stackrel{\mathsf{R}}{=} \Sigma \wedge \sigma \stackrel{\mathsf{L}}{=} \Sigma \wedge \sigma \stackrel{\mathsf{T}}{=} \Sigma
\end{aligned}
$$

二元关系 $\sigma \overset{MRLT}{=\!=} \Sigma$ 的意思是: 等式左右两边的弱机器状态 σ 和强机器状态 Σ 的计时器，寄存器文件和锁资源状态必须相同。对于内存来说，尽管弱机器状态的内存中的历史记录可能包含多个写动作，但是在 DRF 程序下，这些写动作一定有一个全局顺序，并且只有最近的写动作才可能被读到。而该写动作的值必须要和强机器状态内存所对应的变量的值相同。

我们同时还定义了 $\mathit{buff}(\sigma)$，这是事件缓存和所有线程局部的重放缓存的合集。

$$\mathit{buff}(\sigma) \overset{\text{def}}{=} \{e \mid e \in \sigma.\mathrm{b} \vee \exists i.\, e \in \sigma.\mathit{tq}(i).\mathit{rb}\}$$

细心的读者可能已经注意到了，在绝大部分情况下，弱机器状态 σ 的事件缓存和重放缓存并不为空，意思是说在任意的时间点它可以执行缓存中的某个或某些事件到达另一个状态。因此我们需要定位弱机器状态和强机器状态之间约等于的关系，即我们说一个弱机器状态 σ 和一个强机器状态 Σ 是约等于关系当且仅当存在这么一个弱机器状态 σ' 是 σ 把它所有缓存清空执行后达到的状态且 $\sigma' \overset{MRLT}{=\!=} \mathrm{State}$, 并且无论 σ 执行它的缓存事件是按什么顺序，最后当所有缓存都清空以后，它总是达到 σ' 这个状态。形式化定义如下：

$$\begin{aligned}
(P,\sigma) \twoheadrightarrow (P,\sigma') &\overset{\text{def}}{=} (P,\sigma) \xrightarrow{lab} (\mathrm{P},\sigma') \\
&\qquad \wedge lab.\mathrm{a} \in \{\mathrm{a} \mid \mathrm{a} = \mathbf{BufA}\ _\} \\
(\mathrm{P},\sigma) \Downarrow (P,\sigma') &\overset{\text{def}}{=} (P,\sigma) \twoheadrightarrow^* (P,\sigma') \wedge \mathit{buff}(\sigma') = \emptyset \\
(\mathrm{P},\sigma) \approx (P,\Sigma) &\overset{\text{def}}{=} (\exists \sigma'.\, (P,\sigma) \Downarrow (P,\sigma') \wedge \sigma' \overset{MRLT}{=\!=} \Sigma) \\
&\qquad \wedge (\forall \sigma_1, \sigma_2.\, (P,\sigma) \Downarrow (P,\sigma_1) \wedge (P,\sigma) \Downarrow (P,\sigma_2) \\
&\qquad\qquad \Rightarrow \sigma_1 = \sigma_2)
\end{aligned}$$

为了描述方便，我们使用 $(\mathrm{P},\sigma) \twoheadrightarrow^* \xrightarrow{lab} (\mathrm{P}',\sigma')$ 来表达：存在这么一个 σ'' 使得 $(\mathrm{P},\sigma) \twoheadrightarrow^* (\mathrm{P},\sigma'') \xrightarrow{lab} (\mathrm{P}',\sigma')$。同样 $(\mathrm{P},\sigma) \rightarrow^* \xrightarrow{lab} (\mathrm{P}',\sigma')$ 意味着存在这么一个 σ'' 使得 $(\mathrm{P},\sigma) \rightarrow^* (\mathrm{P},\sigma'') \xrightarrow{lab} (\mathrm{P}',\sigma')$。

定义 3.4.6. *对于任意给定的程序 P, OHMM 机器状态 σ, SCM 机器状态 Σ, 若 $\mathit{buff}(\sigma) = \emptyset$ 并且 $(P,\sigma) \approx (P,\Sigma)$, 那么我们可以定义关系二元 $\sim_{\mathrm{P}}^{\sigma\Sigma}$ 作用在弱机器配置和强机器配置上，如下所示:*

- $(P,\sigma) \sim_P^{\sigma\Sigma} (P,\Sigma)$ *总是成立。*
- $(\mathrm{P}',\sigma') \sim_{\mathrm{P}}^{\sigma\Sigma} (\mathrm{P}',\Sigma')$ *当且仅当存在这么一个弱执行序列:*

$$(\mathrm{P},\sigma) \twoheadrightarrow^* \xrightarrow{lab_0} (\mathrm{P}^1,\sigma^1) \cdots \twoheadrightarrow^* \xrightarrow{lab_{n-1}} (\mathrm{P}^n,\sigma^n) = (\mathrm{P}',\sigma')$$

使得:

$$(\mathrm{P},\Sigma) \xrightarrow{lab_0} (\mathrm{P}^1,\Sigma^1) \ldots \xrightarrow{lab_{n-1}} (\mathrm{P}^n,\Sigma^n) = (\mathrm{P}',\Sigma')$$

是一个合法 (符合操作语义的) 的强执行序列，并且我们有对于任意的 i, 若 $i \in (0,n]$ 则有 $(P^i,\sigma^i) \approx (P^i,\Sigma^i) \wedge lab_{i-1}.a \notin \{\mathrm{a} \mid \mathrm{a} = \mathbf{BufA}\ \_\}$。

接下来我们就能证明，符合一定条件下，强机器状态可以完全产生跟弱机器状态一样的行为:

引理 3.4.1 (Simulation). *如果* (P^0, Σ^0) *是无数据竞争的并且*

$$\sigma^0 \xlongequal{MRLT} \Sigma^0 \wedge buff(\sigma^0) = \emptyset$$

, 那么对于任意的 $P', \sigma', \Sigma', P'', \sigma'', lab,$ *如果我们有*

$$(P', \sigma') \sim_P^{\sigma^0 \Sigma^0} (P', \Sigma') \wedge (P', \sigma') \twoheadrightarrow^* \xrightarrow{lab} (P'', \sigma'') \wedge lab.a \notin \{a \mid a = \textbf{\textit{BufA}}\ \_\}$$

那么存在 Σ'' *使得*

$$(P', \Sigma') \xrightarrow{lab} (P'', \Sigma'') \wedge (P'', \sigma'') \sim_P^{\sigma^0 \Sigma^0} (P'', \Sigma'')$$

证明. 这段证明我们放在附录 A中。 □

然后，为了证明，对于 DRF 的程序来说，弱机器语义不会产生和强机器状态不一样的行为，我们需要以下的引理：

引理 3.4.2. *如果* (P^0, Σ^0) *无数据竞争,* $\sigma^0 \xlongequal{MRLT} \Sigma^0, buff(\sigma^0) = \emptyset,$ *并且对于任意的* $P', \sigma',$ *若*

$$(P^0, \sigma^0) \twoheadrightarrow^* \xrightarrow{lab} (P', \sigma') \wedge lab.act \notin \{a \mid a = \textbf{\textit{BufA}}\ \_\}$$

那么存在 Σ' *使得* $(P', \sigma') \sim_{P^0}^{\sigma^0 \Sigma^0} (P', \Sigma')$。

证明. 我们假设从 (P^0, σ^0) 到 (P', σ') 的弱执行序列如下:
$\mathscr{W} = (P^0, \sigma^0) \twoheadrightarrow^* \xrightarrow{lab_0} (P^1, \sigma^1) \cdots \twoheadrightarrow^* \xrightarrow{lab_{n-1}} (P^n, \sigma^n)$ 并且有 $(P^n, \sigma^n) = (P', \sigma')$ 以及 $lab_{n-1} = lab$。
因此我们可以在 n 上做归纳假设:

- Base: $n = 0$. 根据引理3.4.1, 我们知道 $(P^0, \sigma^0) \sim_{P^0}^{\sigma^0 \Sigma^0} (P^0, \Sigma^0)$。
- Induction: $n > 0$. 根据归纳假设，存在 Σ'' 使得

  $$(P^{n-1}, \sigma^{n-1}) \sim_P^{\sigma^0 \Sigma^0} (P^{n-1}, \Sigma'')$$

  由于我们有

  $$(P^{n-1}, \sigma^{n-1}) \twoheadrightarrow^* \xrightarrow{lab_{n-1}} (P^n, \sigma^n)$$

  我们可以运用引理3.4.1得到存在 Σ' 使得

  $$(P^{n-1}, \Sigma'') \xrightarrow{lab} (P^n, \Sigma') \wedge (P^n, \sigma^n) \sim_P^{\sigma^0 \Sigma^0} (P^n, \Sigma')$$

  然后我们可以得出结论。

□

推论 3.4.1. *如果* (P, Σ) *无数据竞争, 并且* $\sigma \xlongequal{MRLT} \Sigma \wedge buff(\sigma) = \emptyset,$ *那么对于任意的* P', σ' *若* $(P, \sigma) \rightarrow^* (P', \sigma'),$ *则存在* Σ' *使得* $(P, \Sigma) \rightarrow^* (P', \Sigma')$ *并且* $(P', \sigma') \approx (P', \Sigma')$。

证明. 我们对 $(P,\sigma) \to^* (P',\sigma')$ 进行分情况讨论：

- 如果有 $(P,\sigma) \to^* \xrightarrow{lab} (P',\sigma')$。应用引理 3.4.2, 存在 Σ' 使得 $(P',\sigma') \sim_p^{\sigma\Sigma} (P',\Sigma')$。根据 $\sim_p^{\sigma\Sigma}$ 的定义, 我们有

$$(P,\Sigma) \to^* (P',\Sigma') \wedge (P',\sigma') \approx (P',\Sigma')$$

- 如果有 $(P,\sigma) \to^* \xrightarrow{lab} (P',\sigma'') \twoheadrightarrow^* (P',\sigma')$. 运用引理 3.4.2 我们知道存在 Σ'' 使得 $(P',\sigma'') \sim_p^{\sigma\Sigma} (P',\Sigma'')$。根据 $\sim_p^{\sigma\Sigma}$ 的定义, 我们有

$$(P,\Sigma) \to^* (P',\Sigma'') \wedge \sigma'' \approx \Sigma''$$

  由于 $(P',\sigma'') \twoheadrightarrow^* (P',\sigma)$, 我们因此能得出 $\sigma' \approx \Sigma''$。

□

定理 3.4.1 (DRF-Guarantee). *对于任意的 P, σ 以及 Σ, 若 (P,Σ) 无数据竞争, $\sigma \xrightarrow{MRLT} \Sigma$ 并且 $buff(\sigma) = \emptyset$, 那么以下成立:*

- *若 $(P,\sigma) \longmapsto^* (\mathbf{skip},\sigma')$ 并且 $buff(\sigma') = \emptyset$, 那么存在 Σ' 使得 $(P,\Sigma) \longrightarrow^* (\mathbf{skip},\Sigma')$ 并且 $\sigma' \xrightarrow{MRLT} \Sigma'$。*
- *如果 $(P,\Sigma) \longrightarrow^* (\mathbf{skip},\Sigma')$, 那么存在 σ' 使得 $(P,\sigma) \longmapsto^* (\mathbf{skip},\sigma')$, $buff(\sigma') = \emptyset$ 并且 $\sigma' \xrightarrow{MRLT} \Sigma'$。*

这个定理说的是，从相关的弱机器初始状态开始，如果程序是 DRF 的并且在我们的语义下执行，最后弱机器状态变成 σ'，那么从相应的强机器状态开始，根据顺序一致性的语义执行程序，强机器状态应当到达某个 Σ', 使得 $\sigma' \xrightarrow{MRLT} \Sigma'$; 反之亦然。

证明. DRF-Guarantee 性质的前半部分显然成立，因为这是推论 3.4.1的一个特例。而后半部分我们需要证明弱语义可以模拟强语义的任何行为。这个结论是平凡的，因为在弱语义中，每个事件的发出都是根据交错语义。我们可以很简单的通过立即执行一个事件在它被放入事件缓存之后，并且永远不重放它来模拟强抽象机器。所以在这样的条件下，弱语义应当可以精确的模拟强语义的任何行为。 □

3.5　程序变换算法的正确性

跟随 Ševčík 和 Aspinall [32] 的工作, 我们在这一节中研究一些简单的程序变换在 OHMM 下的正确性。一个程序变换的正确性是说，目标程序（经过程序变换后产生的新程序）的所能产生的行为是源程序所能产生的行为的子集，也就是说目标程序不会比源程序产生更多的行为。程序变换应当在任何上下文中都成立。

如图 3.11所示, 我们使用了和 Ševčík 以及 Aspinall [32] 他们工作里相同的程序变换集合来对 OHMM 上的正确性进行讨论，除了 trace-preserving 变换和外部

Transformation	SC	JMM	JMM-Alt	OHMM
Reordering normal memory accesses (R-RR, R-WW, R-WR, R-RW)	×	×	√	√
Redundant read after read elimination(E-RAR)	√	×	×	√
Redundant read after write elimination(E-WAR)	√	√	√	√
Irrelevant read elimination (E-IR)	√	√	√	√
Irrelevant read introduction	√	×	×	√
Redundant write before write elimination(E-WBW)	√	√	√	√
Redundant write after read elimination(E-WAR)	√	×	×	√
Roach-motel reordering(RoachMotel-L, RoachMotel-U)	×(√ for locks)	×	×	√

图 3.11: 程序变换的正确性

动作调序变换。我们去掉前者的原因是因为我们没有用 trace 语义来定义我们的模型，而去掉后者是因为我们的语言不会产生外部事件。

图 3.11同时展示了程序变换在顺序一致性模型上，JMM, JMM-Altt [26] 以及 OHMM 上的正确性。前面三者的结果是直接摘选自 [32]。我们在其基础上额外提供了 OHMM 的正确性。如图所示，所有我们在这边讨论的程序变换在我们的模型下都是合法的，因此 OHMM 能够允许更多的优化（值得注意的是，我们这里研究的程序变换更多的是基于语法定义, 参见图 3.12和图 3.13. 而 [32] 中更多的是基于语义定义，因此可能会比我们的更加通用）。值得注意的是，所有变换只能作用在普通变量上。因为 volatile 变量的读写一般是用来同步线程的，就像 lock/unlock 指令那样，因此我们认为它们不应当被编译器所修改。

3.5.1　程序变换

我们把程序变换分为两类，删除冗余语句和调整语句顺序。而程序变换中有一种类型是冗余读操作引入，这个变换类型不属于我们上面说的两类，但是它可以被定义成冗余读操作删除的反变换 a（下文中的 Rule E-IR ），因此我们把它放入下文删除冗余语句的小节中一并进行说明。

3.5.1.1　删除变换

我们所考虑的删除变换 $\xrightarrow{e}$ 的定义如图 3.12中所示：

- 删除冗余的读操作 (Rule E-RAR 和 Rule E-RAW)。我们知道，从内存中读取的速度大大慢于直接从寄存器中读取的速度。因此，这两条变换操作是用来将发生在对同一个内存地址的读/写操作后面的读内存操作替换成相应的从前者所使用的寄存器中读取的操作。这个程序变换在编译器设计中经常用于公共子表达式删除或者常量传播优化。在下面的例3.5.1中, 我们使用 Rule E-RAR 作用在左边的程序上来获得右边经过变换后的程序。我们可以看到，右边的程序使用存在寄存器 r_1 中的上一次读取 x 的值来替换重新从 x 中读取。事实上，我们可以观察到，经过这次变换以后，编译器就能作更多的后续优化，比如编译器就能判断 r_2 和 r_1 的值恒相等，因

此 IF 语句的条件表达式恒成立，可以将对 y 的写操作提出条件语句外进行调序等操作。

例 3.5.1 (E-RAR)**.**

$$\begin{array}{l} r_1 := x; \\ r_2 := x; \\ \mathit{if}(r_1 == r_2) \\ \quad y := 1; \end{array} \xrightarrow{e} \begin{array}{l} r_1 := x; \\ r_2 := r_1; \\ \mathit{if}(r_1 == r_2) \\ \quad y := 1; \end{array}$$

- 删除紧跟在读操作后面的冗余的写操作 (rule E-WAR)。如例3.5.2中所示，对 x 的写操作 $x := r_1$ 可以被删除，因为在它之前的读操作刚刚把 x 的值存与于 r_1 中，而 r_1 是寄存器变量，意味着外界的环境不能对 r_1 的值产生任何影响。因此 $x := r_1$ 的值不会改变 x 中原有的值。Vafeiadis 等人指出 E-WAR 这条变换在同样基于 Happens-before 的 C11 模型下并不正确并且列出了一条反例 ([33] 中的图 9)。然而，这个反例在我们模型中并不成立，原因在于在我们的模型中对非 volatile 变量的读写操作比在 C11 模型中的 Relaxed Memory Order(RLX) 要弱，因此我们对于来自于不同线程对同一个内存位置的写操作，我们并不存在全序 mo 关系。

例 3.5.2 (E-WAR)**.**

$$\begin{array}{l} r_1 := x; \\ x := r_1; \end{array} \xrightarrow{e} \; r_1 := x;$$

- 删除紧跟在写操作后面的冗余的写操作 (Rule E-WBW). 如例3.5.3中所示，这种变换经常用于 peephole 优化中 (Aho 等，1986)。

例 3.5.3 (E-WBW)**.**

$$\begin{array}{l} x := 1; \\ x := 3; \end{array} \xrightarrow{e} \; x := 3;$$

- 删除在一个对寄存器赋值前的冗余的读操作 (Rule E-IR). 举例来说 $r_1 := x; r_1 := 1$ 可以变换成 $r_1 := 1$, 因为寄存器 r_1 在读取完内存单元 x 的值后，还没有被使用就立即被值 1 所覆盖，因此这个读操作 $r_1 := x$ 是冗余的。编译器经常会使用这种变换来实现死代码删除优化。

- 如我们之前所述，冗余的读操作引入是 Rule E-IR 的逆变换。这个变换看起来好像并不算是优化，因为比起变换前反而引入了读操作，但是现代的处理器硬件却经常通过这个变换来进行投机读取。比如下面例3.5.4中所示，我们通过对左边的程序引入了一个对 x 的冗余读操作 $r_2 := x$ 得到中间的程序。然后编译器就知道，对于中间这个程序来说，无论 IF 语句的条件表达式为真否，语句 $r_2 := x$ 总是要执行的，因此，这个程序就和右边的程序等价。这样，就等于编译器投机读取了 x 的值。

$$\textit{(SeqContext)} \qquad \mathbb{E} ::= [\,] \mid \mathsf{E}; \mathsf{C}$$

$$\frac{}{C \xrightarrow{e} C} \qquad \frac{tid.C \xrightarrow{e} tid.C'}{\mathbb{P}[tid.C] \xrightarrow{e} \mathbb{P}[tid.C']} \qquad \frac{C \xrightarrow{e} C'}{tid.\mathbb{E}[C] \xrightarrow{e} tid.\mathbb{E}[C']}$$

$$\frac{}{r_1 := \mathsf{x}; r_2 := \mathsf{x}; \xrightarrow{e} r_1 := \mathsf{x}; r_2 := r_1;} \text{(E-RAR)}$$

$$\frac{}{\mathsf{x} := r_1; r_2 := \mathsf{x}; \xrightarrow{e} \mathsf{x} := r_1; r_2 := r_1;} \text{(E-RAW)}$$

$$\frac{}{r := \mathsf{x}; \mathsf{x} := r; \xrightarrow{e} r := \mathsf{x};} \text{(E-WAR)}$$

$$\frac{}{\mathsf{x} := r_1; \mathsf{x} := r_2; \xrightarrow{e} \mathsf{x} := r_2;} \text{(E-WBW)}$$

$$\frac{r \notin \mathit{UseR}(\mathsf{E})}{r := \mathsf{x}; r := \mathsf{E}; \xrightarrow{e} r := \mathsf{E};} \text{(E-IR)}$$

图 3.12: 删除冗余变换

例 3.5.4 (I-IR)**.**

$$\begin{array}{l} \textit{if}\,(\mathsf{r}_1 == 1) \\ \quad \{\mathsf{r}_2 := \mathsf{x}; \\ \quad\ \mathsf{y} := \mathsf{r}_2\} \\ \mathsf{r}_2 := 1; \end{array} \xrightarrow{e} \begin{array}{l} \textit{if}\,(\mathsf{r}_1 == 1) \\ \quad \{\mathsf{r}_2 := \mathsf{x}; \\ \quad\ \mathsf{y} := \mathsf{r}_2;\} \\ \textit{else}\ \mathsf{r}_2 := \mathsf{x}; \\ \mathsf{r}_2 := 1; \end{array} \rightleftharpoons \begin{array}{l} \mathsf{r}_2 := \mathsf{x}; \\ \textit{if}\,(\mathsf{r}_1 == 1) \\ \quad \{\mathsf{y} := \mathsf{r}_2;\} \\ \mathsf{r}_2 := 1; \end{array}$$

3.5.1.2　调序变换

调换紧邻的两个指令的执行顺序也是一种重要的程序变换，其中大体包括三类：调换两个互相没有依赖性关系的非同步指令; 调换 Lock 指令和它之前的非同步指令; 调换 Unlock 指令和它之后的非同步指令。调序优化经常用于硬件或者编译器的循环优化中。值得注意的是 Manson 等人 [20] 宣称对非同步指令的调序变换在 JMM 中是正确的，然而 Cenciarelli 等人 [13] 找到一个反例，然后 Aspinall 等 [26] 修复了这个 bug, 并把修改之后的 JMM 称之为 JMM-Alt（我们在前面讲解例子的时候曾经提到过）。我们的内存模型可以相当好的支持这种变换，如图 3.11中所示，并且我们在附录 B中证明了这一点。更重要的是，我们的模型还支持 Roach-调序变换而不论 JMM 还是 JMM-Alt 都不能做到这一点。如我们在节 3.3中的例3.3.6指出的那样，支持这种变换会让 roach motel 程序的行为更加的符合直观: 即对一个程序增加更多的同步语句会导致程序更少的行为可能甚至死锁（JMM 和 JMM-Alt 中反而会引入更多的行为可能性，这一点看起来非常的违反直观）。

在我们的语言中，调序变换 $\xrightarrow{s}$ 可以定义成如图 3.13中所示，包括：

$$\frac{}{C \xrightarrow{s} C} \qquad \frac{tid.C \xrightarrow{s} tid.C'}{\mathbb{P}[tid.C] \xrightarrow{s} \mathbb{P}[tid.C']} \qquad \frac{C \xrightarrow{s} C'}{tid.\mathbb{E}[C] \xrightarrow{s} tid.\mathbb{E}[C']}$$

$$\frac{r_1 \neq r_2}{r_1 := x_1; r_2 := x_2; \xrightarrow{s} r_2 := x_2; r_1 := x_1;} \text{ (R-RR)}$$

$$\frac{x_1 \neq x_2}{x_1 := r_1; x_2 := r_2; \xrightarrow{s} x_2 := r_2; x_1 := r_1;} \text{ (R-WW)}$$

$$\frac{x_1 \neq x_2 \quad r_1 \neq r_2}{x_1 := r_1; r_2 := x_2; \xrightarrow{s} r_2 := x_2; x_1 := r_1;} \text{ (R-WR)}$$

$$\frac{x_1 \neq x_2 \quad r_1 \neq r_2}{r_1 := x_1; x_2 := r_2; \xrightarrow{s} x_2 := r_2; r_1 = x_1;} \text{ (R-RW)}$$

$$\frac{}{\iota_n; \textbf{lock}\ l; \xrightarrow{s} \textbf{lock}\ l; \iota_n;} \text{ (RoachMotel-L)}$$

$$\frac{}{\textbf{unlock}\ l; \iota_n; \xrightarrow{s} \iota_n; \textbf{unlock}\ l;} \text{ (RoachMotel-U)}$$

图 3.13: 调序变换

- 调换两个没有数据依赖性并且没有发生冲突的内存访问 (Rule R-RR, R-WW, R-WR 和 R-RW)。
- 调换 Lock 指令和它之前的非同步指令 (Rule RoachMotel-L)。
- 调换 Unlock 和它之后的非同步指令 (Rule RoachMotel-U)。

我们需要指出的是，程序变换的正确性只是需要保证变换的目标程序不会比变换前的源程序拥有更多的行为。因此，源程序确实有可能会比目标程序具有更多的行为，下面的例子说明了这一点：

例 3.5.5.

$$\begin{array}{l} r_1 := x; \\ x := r_1; \end{array} \;\Big\|\; \begin{array}{l} r_2 := x; \\ x := 2; \end{array} \quad \xrightarrow{e} \quad r_1 := x; \;\Big\|\; \begin{array}{l} r_2 := x; \\ x := 2; \end{array}$$

在例3.5.5中，目标程序是运用 Rule E-WAR 作用在源程序左边的线程上得到的。在源程序中 r_2 可以读到值 2(被 $x := r_1$ 这个写操作所写入的值)，但是在目标程序中却不可能，因为在目标程序中，唯一的写操作 $x := 2$ 按照程序顺序是在 $r_2 := x$ 之后的，根据我们对 *visible* 的定义 (参见图 3.5)，r_2 只能读到初始值 0。

3.5.2　变换的正确性

在这一节中，我们将给出关于程序变换在 OHMM 上正确性的简要概述。更多的细节参见附录 B。

因为我们的语言没有输出语句，所以当我们比较程序之间的行为，我们事实上是比较它们从同一个机器状态开始所能到达的终态。下面我们定义两个机器状态 σ 和 σ' 的观测等价性 $\sigma \overset{obsv}{=\!=} \sigma'$:

$$
\begin{aligned}
&\sigma \overset{\mathrm{r}}{=} \sigma' \quad \overset{\text{def}}{=} \sigma.tq.rf = \sigma'.tq.rf \\
&\sigma \overset{\mathrm{l}}{=} \sigma' \quad \overset{\text{def}}{=} \sigma.\mathrm{L} = \sigma'.\mathrm{L} \\
&\sigma \overset{\mathrm{b}}{=} \sigma' \quad \overset{\text{def}}{=} \mathit{buff}(\sigma) = \mathit{buff}(\sigma') = \emptyset \\
&\sigma \overset{\mathrm{m}}{=} \sigma' \quad \overset{\text{def}}{=} \mathsf{dom}(\sigma.m) = \mathsf{dom}(\sigma'.m) \\
&\qquad\qquad \wedge \forall \mathrm{x} \in \mathsf{dom}(\sigma.m).\, (\sigma.m(\mathrm{x}) = \sigma'.m(\mathrm{x}) \\
&\qquad\qquad\quad \vee \forall \mathrm{i}.\, \mathit{visibleV}(\mathrm{i}, \sigma.\mathrm{t}, \sigma.m(\mathrm{x})) = \mathit{visibleV}(\mathrm{i}, \sigma'.\mathrm{t}, \sigma'.m(\mathrm{x})))
\end{aligned}
$$

$$
\sigma \overset{obsv}{=\!=} \sigma' \overset{\text{def}}{=} \sigma \overset{\mathrm{r}}{=} \sigma' \wedge \sigma \overset{\mathrm{b}}{=} \sigma' \wedge \sigma \overset{\mathrm{m}}{=} \sigma' \wedge \sigma \overset{\mathrm{l}}{=} \sigma'
$$

where $\mathit{visibleV}(\mathrm{i}, \mathrm{t}, \mathrm{h}) \overset{\text{def}}{=} \{wv.v \mid wv \in \mathrm{h}.\mathit{visible}(\langle \mathrm{i}, \mathrm{t} \rangle, wv, \mathrm{h})\}$

注意我们并不要求 σ 和 σ' 完全一模一样，因为这个对于基于 histroy 的内存来说是没有必要的过于强的约束。我们说，两个机器状态拥有相同的内存 (*i.e.*, $\sigma \overset{\mathrm{m}}{=} \sigma'$) 当且仅当任何后续的读操作可以看到相同的值的集合 (请回忆一下我们在图 3.5中对 $\mathit{visible}(ts, wv, \mathrm{h})$ 的定义)。

我们使用 $P \xrightarrow{\mathrm{t}} P'$ 代表从 P 到 P' 的程序变换，包括删除变换和调序变换，即 $\xrightarrow{\mathrm{t}} \overset{\text{def}}{=} \xrightarrow{\mathrm{e}} \cup \xrightarrow{\mathrm{s}}$。同时，我们还定义 $\xrightarrow{\mathrm{t}}{}^*$ 是 $\xrightarrow{\mathrm{t}}$ 的传递反射闭包。

定理 3.5.1 (Validity of Transformation). *对于任意的 P, σ 以及 P', 若*

$$
P \xrightarrow{\mathrm{t}}{}^* P' \wedge (\mathrm{P}', \sigma) \longrightarrow^* (\mathbf{skip}, \sigma') \wedge \mathit{buff}(\sigma') = \emptyset
$$

则存在 σ'' 使得

$$
(\mathrm{P}, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma'') \wedge \sigma' \overset{obser}{=\!=} \sigma''
$$

定理 3.5.1意思是，从同一个初始化状态开始，如果一个经过若干变换的目标程序能够到达一个最终状态 σ', 那么源程序应当能够到达某个最终状态 σ'', 使得 σ' 和 σ'' 观测等价。

为了证明该定理，我们可以先通过建立目标程序和源程序的模拟关系来分别证明每个变换的正确性，然后根据 $\overset{obser}{=\!=}$ 的传递性，我们可以知道由任意变换的组合都是正确的，即证明了该定理。

3.6　相关工作

在这一节中我们将介绍国外国内的相关工作。我们知道，已经由很多的关于弱内存模型的工作，我们在这边只讨论和我们关系比较近的工作。

Yang 等人 [34] 提出了一个通过 Uniform Memory Model(UMM) 建立的 Java 线程语义模型。他们的工作和我们在许多方面类似，比如两者都是基于操作语

义，都是定义在抽象机的执行基础上。在 UMM 中，抽象机拥有线程局部的指令缓存，和我们的模型中全局的事件缓存类似，都能允许语句或者指令的调序执行。UMM 同时拥有一个全局的指令缓存用来保存所有之前的内存写操作，这一点和我们基于历史记录的内存的作用类似。然而，UMM 没有重放机制，所以很多重要的程序变换，比如例3.1.4它就不能支持。

在 UMM 的工作基础上，有许多工作都试图去修补早期的 Java 内存模型，直到 Manson 等人 [12] 提出了当前的 Java 内存模型，JMM, 该模型现在已经被加入 Java 规范的第 17 章中作为并行编程中相当重要的一节。如同 Manson 在他们文章中指出的那样，绝大部分对早期 Java 内存模型进行修补的工作都不能够支持很多相当重要的弱序执行。

而从当前的 JMM 提出后，又有许多工作来对其进行形式化并且证明它满足 DRF-Guarantee [21, 26, 35]。他们的形式化工作严谨并且缜密，对理解 JMM 和发现其中的 bug 有相当重要的帮助。然而他们都是从原始版本的公理语义出发，而不是使用操作语义定义。

Saraswat 等 [36] 提出了一种新的弱内存模型，RAO. 这个模型通过程序变换来弱化行为约束。程序变换的种类被精心挑选，以用来避免 out-of-thin-air 行为的出现。如同 JMM 一样，RAO 模型禁止因果测试例中的 case 5 和 case 10, 以及在 Manson 等人的工作中 [12] 中第 8 节所提出争议性例子。而这些例子，在我们的模型中，都是允许的。

Cenciarelli 等人 [13] 使用了结合了操作语义，公理语义以及标记技术的语义框架来形式化 JMM。他们使用一种称之为 “configuration theory” 的方法来具体定义事件的依赖性。他们允许在相应的代码执行之前就将事件加入 configuration 中来实现投机执行。然后在接下来的执行中需要证明之前的投机执行是否**满足**一定的条件即是否合法，如果不是的话，则抛弃整条预测执行后的执行路径。我们则不是使用这样的直接投机来模拟投机执行。相反，我们的重放机制允许我们将一段代码运行若干次，而该段代码第一次执行的结果可以视为投机预测，用来决定接下来的执行路径。这样的投机预测由于仍然是基于操作语义以及当前的机器状态的，因此总是合法的，我们并不需要额外的证明投机是否合法。

Jagadeesan 等人 [37] 提出了一种基于 generative 的操作语义来定义弱内存模型。他们的模型在内存中保存了所有的写事件，就和我们使用历史记录一样。和 Cenciarelli 等人 [13] 的工作类似，他们通过不确定性的预测内存的值来实现投机执行。同样，这样的预测也需要后续的证明来验证是否合法。如同我们上面解释过的那样，这和我们的重放机制不同。尽管这个模型允许许多被 JMM 所禁止的无锁程序的行为，但是它还是不允许那些有争议的被 JMM 所禁止的行为 (如例 3.3.4)。这些例子在我们的模型中都是允许的，所以看起来我们的重放机制比他们支持投机的方法更加广泛。

Sarkar 等人 [38] 提出了基于 Power 多核处理器的语义模型。他们的 “storage

subsystem” 和我们的内存以及缓存机制类似。他们通过允许在一条指令被 commit 之前重做这条语句来实现投机，看起来和我们的重放机制很像。但是他们的指令只能在 commit 之前重做，意味这其他线程将不能看到这条指令的效果，直到它被 commit。这一点和我们的重放机制不同，我们的重放机制在第一次执行完毕之后就能被其他线程所看到。Power 内存模型的约束要强于我们的模型和 JMM。

Boudol 和 Pertri [39] 和我们的工作相同的地方在于他们提出了同样基于操作语义的弱内存模型。他们只对写操作进行缓存，同时他们也使用模拟方法证明了该模型具有 DRF-guarantee 的性质。但他们模型的约束要强于我们的 OHMM, 并且不能支持对投机执行的抽象。Demange [18] 等人的工作则是提出了同样是对写操作进行缓存并且同时是 JMM 模型行为子集的弱内存模型。他们分别使用了操作语义和公理语义定义出该模型，并且巧妙地证明了两种方法所定义出的模型的等价性。

3.7　总结和弱点探讨

在这一章中的前面几节，我们主要提出了一种基于 Happens-before 的使用操作语义定义的内存模型。为了支持编译优化或硬件优化中的投机执行，我们引入了重放机制，可以让我们将一段代码重复运行若干次来模拟投机执行中的静态分析和后续的程序调序优化。我们希望我们使用的基于状态转移的操作语义可以让我们的模型更好的被程序员理解。并且我们的模型满足我们在节 1.1中提到的三条标准。即我们的模型满足 DRF-Guarantee, 并且禁止了原生的 happens-before 模型中有害的 out-of-thin-air 行为的出现。另一方面，我们的模型的约束害做到尽可能的弱。对于无锁程序，我们的模型允许那些 JMM 和 JMM 的变种 (比如 Jagadeesan 等提出的 [37]) 所禁止但是无害的行为的发生。除此之外，我们在节 3.3中还特别讨论了那些违反直观认识的 JMM 中的特性。我们还研究了常见的简单程序变换在 OHMM 下的正确性，这些程序变换有些在 JMM 中是不正确的，但是我们的模型却能很好的支持它们。除了上述的理论工作之外，我们还对论文中的定理部分进行了机器证明，并且提供了一个符合我们模型的解释器来解释执行程序 [31]。

弱点：我们的模型所采用的语言是一个 toy language, 缺少许多现代并发语言的特性比如线程创建，线程等待，常量域初始化，对象初始化等等。因此我们的模型还是在这方面还是比较初步的，要完全取代 JMM 或者成为 Java 可选的内存模型还为时尚早。另一方面，我们在工作中没有对我们的模型如何在现有的处理器架构上能够有效的实现进行探讨和证明。这方面目前只是一个初步的设想，粗略的说，由于在我们模型中，volatile 变量的读/写操作行为和 C11 模型中的 Load Require/Store Release 的 atomic 操作类似, 跟随 McKenney 和 Silver 的工作 [23]，在 Power 架构上我们可以使用 “ld; cmp; bc; isync”/ “lwsync; st” 来

对 volatile 变量的读/写进行实现; 而在 X86-TSO 上，我们可以使用 “ld; mfence”/ “mfence; st” 来实现，但是这可能会比在我们模型上的约束要强一些。

第四章　从 OHMM 到 TSO 弱内存模型

我们知道，TSO 模型已经被用作 X86 和 SPARC-TSO 处理器族的模型基础，并且在一些高级语言中也正在被提案作为其内存模型的基础, 这是一个被广泛使用的内存模型。回忆我们在图 2.2中提到的 TSO 抽象机以及对 TSO 内存模型的介绍，对比与我们在第三章中提出的 OHMM 模型，直观上的认识是 TSO 内存模型是一个约束比 OHMM 强很多的弱内存模型，即 OHMM 允许出现的行为比 TSO 多。而我们在本章中将具体论证这一点, TSO 内存模型所允许的程序行为是 OHMM 的一个子集。我们通过先对 OHMM 模型加上一些约束，构建出一个中间内存模型 OHMM-TSO, 然后验证出 TSO 内存模型是 OHMM-TSO 内存模型行为的子集。而因为我们知道 OHMM-TSO 是通过在 OHMM 上加入约束所得，即 OHMM-TSO 是 OHMM 内存模型行为的子集，故我们最终能推导出，TSO 是 OHMM 的子集。

4.1　OHMM-TSO 模型

我们通过对 OHMM 模型加入约束来得到 OHMM-TSO 模型。我们在 OHMM 中提到过，我们使用三种方法来尽量弱化我们的内存，即，事件缓存，基于历史记录的内存以及重放机制。事实上，我们只需要基于历史记录的内存，就能够完全模拟 TSO 模型的行为。具体来说：

1. OHMM-TSO 模型中，事件缓存的大小为 1。也就是说，程序在 OHMM-TSO 中运行时，每条语句被封装成事件以后都需要立即执行, 而不经过事件缓存重新调序。

2. 在 OHMM-TSO 中，所有的重放缓存大小为 0，也就是说，在 OHMM-TSO 中不允许重放机制。

3. 在 OHMM-TSO 中，所有内存单元都是普通变量，不存在 volatile 变量（为了与 TSO 中一致）。

4. 在 OHMM-TSO 中，我们修改了执行读事件所使用的 *visible* 函数 (参见3.1.4)，我们称之为 $visible_{TSO}$（详见下面定义）。而写事件的操作语义保持不变，仍然是将相应的写动作放入历史记录当中。

容易看出，当加入前面三条约束之后，OHMM-TSO 模型会比 OHMM 模型所允许的行为少。我们接下来重点讨论约束 4。

在 OHMM-TSO 中，因为历史记录上保存了所有的写动作，因此我们可以用历史记录来同时模拟 TSO 上的写缓存和内存单元。举个例子来说明，如图 4.1虚

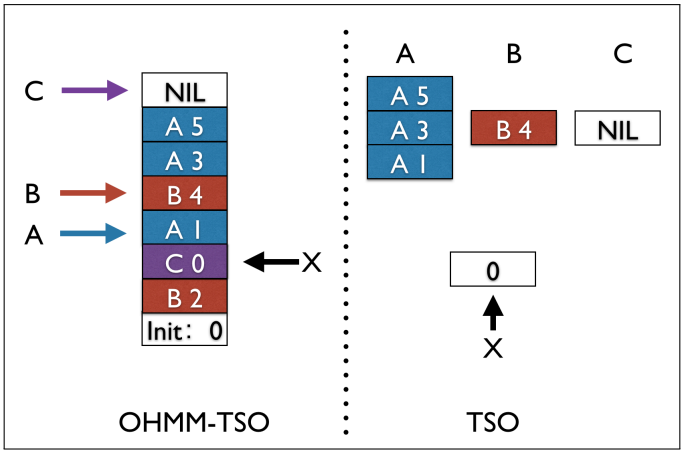


图 4.1: 用 OHMM-TSO 模拟 TSO

线右边所示，在 TSO 模型中，A,B,C 三者分别是三个线程。线程 A 的写缓存中对内存单元 X 有三个写操作，分别是 5，3，1；B 的写缓存中有一个写操作，是 4；而线程 C 的写缓存则为空。则在虚线左边 OHMMTSO 的内存单元 X 的历史记录中，我们加入四个辅助指针来表达出 TSO 此时状态。其中，和线程名称同名的指针，指向的单元表示其对应的 TSO 模型中，写缓存中对该地址的最早的那个写操作。也就是说，意味着在单元 X 历史记录中:

- 位于**指针上面**并且来自于**同一个线程**的写操作都对应位于 TSO 模型中的写缓存中的写操作（比如，A 指针指向 A 1, 在它上面来自于线程 A 的写操作还有 A 3 和 A 5, 故对应的 TSO 中线程 A 的写缓存有 1，3，5 三个写操作）。

- 位于**指针下面**并且来自于**同一个线程**的写操作，都表示 TSO 中已经从缓存中写入内存的写操作。(比如 B 指针指向 B 4, 那么位于它下面的 B 2 就是一个已经被写入内存的写操作。由于 TSO 模型中，内存并不会保存所有的写操作，只会保存最后一个从写缓存中写入内存的值，假设 B 在往内存中写入 2 以后就被 C 往内存中写入的 0 所覆盖，则在 TSO 模型中我们看不到 B 对内存写入 2，如虚线右边所示)。

- 如果指针指向 NIL，说明对应的 TSO 模型中的写缓存为空（比如 C 指针）。

- 和内存单元名同名的指针，指向所对应的 TSO 模型中，该内存目前的值 (比如 X 指针指向 0)。

我们通过该方法，即往 OHMM-TSO 的历史记录中添加指针指向，即可以表达出 TSO 中的内存状态和写缓存状态。

下面是形式化的定义过程：

首先，我们对 OHMM-TSO 中的每个线程添加两个映射函数 f 和 g:

$$\mathrm{f}(\mathrm{x}) = \begin{cases} wv & \text{当内存单元x的历史记录中，该线程指针指向}wv \\ \textbf{nil} & \text{当内存单元x的历史记录中，该线程指针指向}\textbf{nil} \end{cases}$$

$$\mathrm{g}(\mathrm{x}) = \begin{cases} wv & \text{在内存单元x最新的并且来自于该线程的}wv \\ \textbf{nil} & \text{在内存单元x的历史记录中，没有任何来自于该线程的写动作} \end{cases}$$

然后，为 OHMM-TSO 每个内存的历史记录添加一个映射函数 V:

$$\mathrm{V}(\mathrm{x}) = wv \quad \text{当该内存单元中，内存同名指针指向}wv$$

由此，我们可以定义出 $visible_{TSO}$ 函数:

$$visible_{TSO}(\mathrm{x}, tid) \stackrel{\text{def}}{=} wv \qquad \begin{array}{l} \text{当}tid.\mathrm{f}(\mathrm{x}) = wv_1 \text{并且}tid.\mathrm{g}(\mathrm{x}) = wv\text{时} \\ \text{或者当}tid.\mathrm{f}(\mathrm{x}) = \textbf{nil}\text{并且}\mathrm{V}(\mathrm{x}) = wv\text{时} \end{array}$$

其意思是，写动作 wv 对线程 tid 可见，要么是在 x 的历史记录中，所对应的 TSO 模型中 tid 的写缓存最新的写动作是 wv(1)，要么是在 x 的历史记录中，所对应的 TSO 模型中 tid 的写缓存为空，并且所对应的 TSO 模型中内存单元 x 的值是 wv(2)。

由此，我们可以定义出在 OHMM-TSO 中，来自于线程 tid 对内存单元 x 的读事件 e，只能从内存中读到 $visible_{TSO}(\mathrm{x}, tid)$。

对比与 OHMM 内存模型中，$visible$ 函数的定义：

$$\begin{array}{rl} visible(ts, wv, \mathrm{h}) \stackrel{\text{def}}{=} & (wv \prec_{\mathrm{h}}^{\mathrm{hb}} ts \wedge \neg\exists wv'.\, wv \prec_{\mathrm{h}}^{\mathrm{hb}} wv' \wedge wv' \prec_{\mathrm{h}}^{\mathrm{hb}} ts) \\ & \vee\neg(wv \prec_{\mathrm{h}}^{\mathrm{hb}} ts \vee ts \prec_{\mathrm{h}}^{\mathrm{hb}} wv) \end{array}$$

即在 OHMM 中，对于一个读事件来说，它所能读到的写动作要么是 happens-before 于它之前 (3)，要么是和它之间没有任何 Happens-before 关系 (4)。

由于在 OHMM-TSO 中都是普通变量的读写操作，因此来自于不同线程之间的事件之间没有 happens-before 关系。因此在 OHMM-TSO 中，happens-before 关系将退化成程序顺序（见小节 2.2.2）. 因此显然，条件 (1) 是条件 (3) 的子集，即满足于条件 (1）的写动作，一定满足于条件 (3). 而对于条件（2）来说，当它读取的 wv 来自于同一个线程时，条件 (2) 是条件 (1) 的子集，当它读取的 wv 来自于其他线程时，它是条件 (4) 的子集。

由此我们知道，在 OHMM-TSO 中，读事件能够读取到的写动作一定是 OHMM 的子集。因此，我们知道，约束 4 仍然会使得 OHMM-TSO 的行为比 OHMM 的行为少。由此我们可以有下面的引理:

引理 4.1.1. *OHMM-TSO 模型所允许的程序行为，是 OHMM 模型所允许的程序行为的子集。*

证明. 因为约束 1-4 均会使得 OHMM-TSO 的模型的行为变少，因此得证。 □

接下来我们讨论 OHMM-TSO 模型和 TSO 模型的关系。一般来说要证明 TSO 模型是 OHMM-TSO 模型的子集，即 OHMM-TSO 模型能够模拟一切 TSO 模型下可能发生的行为，我们需要建立两者之间的模拟关系，即当某个 TSO 模型状态和 OHMM-TSO 状态等价时，程序在 TSO 状态下执行任意一步走到新的 TSO 状态，则总能从 OHMM-TSO 状态下中找到一步或者若干步走到新的状态，并且 TSO 的新状态和 OHMM-TSO 的新状态仍然等价。然后进行严格的数学证明。而从前文对 OHMM-TSO 的阐述中，我们可以发现这样的模拟关系显而易见，并且本章只是作为前后两章的过渡章节，并不是本文的工作重点，因此我们可以非形式化地这样证明：

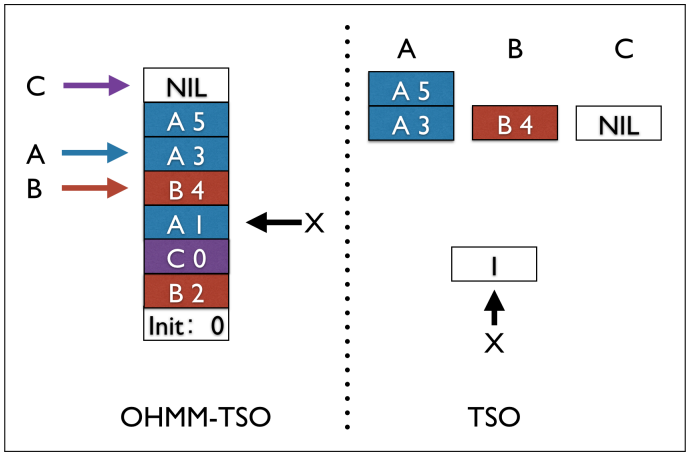
引理 4.1.2. *TSO 模型所允许的程序行为，是 OHMM-TSO 模型所允许的程序行为的子集。*

证明. 我们假设 σ 和 Σ 分别是 TSO 上和 OHMM-TSO 上的机器状态。我们说两者等价即，对于 σ 中任意的内存单元 x 上的值都和 Σ 上 V(x) 的值相等，σ 中任意线程的寄存器的值都和 Σ 上同一个线程的对应的寄存器的值相等，同时 Σ 中的历史记录能够像图 4.1那样模拟出 σ 上内存单元和写缓存的状态。

因此，当 σ 上执行一个读操作的时候到达 σ′ 时，对应的我们在 Σ 上也执行同样的读操作到达 Σ′，并且由于 Σ 的历史记录能够模拟 σ 上内存单元和写缓存的状态，因此两个写操作读到的值是一样的，因此，σ′ 和 Σ′ 仍然等价。

当 σ 上执行把写操作放入写缓存到达 σ′ 时，对应的我们在 Σ 上把写操作放入对应的内存单元的历史记录当中到达新的状态 Σ′。容易证得 σ′ 和 Σ′ 等价。

最后，当 σ 上执行把写缓存中最旧的写操作写入内存并到达 σ′ 时 (假设是线程 A 对内存 x 写入值 n)，对应的我们在 Σ 上，把 V(x) 置为 n, 同时，让线程 A 在历史记录中的同名指针往上移动至下一个对 x 的写操作来到达 Σ′(仍然以图 4.1为例，假设此时在 TSO 模型中，线程 A 把写动作 A 1 写入内存，则在对应的 OHMM-TSO 模型中，我们把 A 指针移动到 A 3。并且，我们把 X 指针移动到 A 1。两者到达的新的状态 σ′ 和 Σ′ 如下图所示), 易证 σ′ 和 Σ′ 仍然等价。



□

定理 4.1.1. *程序在 TSO 下的行为是在 OHMM 下的行为的子集。*

证明. 根据引理4.1.1和引理4.1.2可证。 □

第五章　用于 TSO 模型局部推理的程序逻辑

许多细粒度的并发算法能够在不使用同步原语的情况下实现共享内存的同时访问。尽管已经有各种各样的程序逻辑被设计出来用于证明这些细粒度的同步程序，但是它们中的绝大部分都是基于顺序一致性模型，所以它们并不能对现实世界中那些运行在基于弱内存模型假设下的硬件环境和编译器的程序实现给出正确性的保证。

在本章中，我们提出一种新的用于验证并发程序在 TSO(Total Store Order) 内存模型下行为的程序逻辑。我们的逻辑对冯新宇等人 [7] 的 LRG(Local Rely-Guarantee) 逻辑进行扩展，对其加入了关于 TSO 写缓存的断言, 这可以让我们对 TSO 模型中对外部线程不可见的局部的写缓存的状态进行描述。如同 LRG 一样，我们的程序逻辑支持对细粒度并发具有表达力相当高的 rely/guarantee 推理以及分离逻辑中的局部推理。同时，我们在逻辑上把 TSO 共享内存分为 local 和 shared 两部分，对 TSO 模型进行进一步抽象, 这可以允许我们可以将那些只有对单线程访问的内存单元（逻辑上等同于 local 单元）的写操作不需要经过写缓存直接写入内存。我们用这个逻辑证明了一些有代表性的并发算法在 TSO 上的正确性，包括 Peterson’s lock 算法, Simpson’s Four Slot, Concurrent GCD 算法以及 Optimzed Implementation of Locks 算法。

5.1　语言

直观上，我们的工作中，把标准的 TSO 内存模型 (图 2.2) 再进一步进行抽象和细化成我们称之为 ATSO 的内存模型 (图 5.4)。它和标准的 TSO 的模型差别在于，状态机中每个线程都带有自己的局部内存，对于局部内存的写操作，将不通过写缓存进行缓存，而是直接写入局部内存中。而对于线程之间共享的内存单元的写操作以及读操作等，则是和 TSO 模型一样，通过写缓存进行缓存。同时，我们在 ATSO 抽象机在执行时提供一个对共享内存单元的断言不变式描述 I，如同冯新宇等人的工作 [7] 一样，这个不变式描述了在 ATSO 抽象机执行过程中共享内存的所有可能状态。因此，根据这个不变式，我们就允许 ATSO 的运行过程中动态的将线程局部内存单元移动到共享内存单元或者反过来。这么做的好处是当我们的逻辑的推导规则定义在 ATSO 内存模型上时，我们可以支持细粒度的局部推理，即将那些暂时不会被其他线程读取的内存单元在逻辑上视作局部单元来进行推理。这可以简化我们对程序进行验证时的复杂度。同时，我们会在节 5.5中证明两点: 1. 我们的逻辑在 ATSO 上是可靠的 2.ATSO 模型和 TSO 模型 “等价”。因此，我们可以得出这样的结果：**我们的逻辑在 TSO 内存模型上是可靠的，同时我们又能支持细粒度的局部推理**，这也是本章最大贡献所在。我们的逻辑基于使用类 C 的并发语言定义的 ATSO(图 5.4) 内存模型。

语言的文法定义在图 5.1中定义。**值得注意的是**，这个语言和第三章中所引入的语言基本类似，最大的区别在于由于在 OHMM 中我们关心的是模型本身，因此我们尽可能的简化我们的语言，使用形如 x, y, z 的变量来表示内存上不同的位置而不需要去考虑别名问题。而在这一章节中，由于我们关心的是逻辑是否能够验证程序，因此我们需要在逻辑上考虑别名问题, 所以我们的语言使用形如汇编语言的形式 [l] 来表示内存上地址为 l 的单元。

我们使用 $r_1, r_2, \ldots$ 来代表线程局部的寄存器。语句 $r := [E]$ 和 $[E] := r$ 分别表示对内存单元地址 E 进行读写。值得注意的是，我们的语言中还加入了另一种写语句，我们称之为双写: $\langle [E_1] := E_1', [E_2] := E_2' \rangle$。这种语句由两个写操作组成，其中第二个写操作的地址 E_2 是用作 write-only 辅助历史变量。辅助变量在细粒度的并发验证中是必须的。在顺序一致性模型下的程序逻辑，我们经常用一个原子块 $\langle C; aux := E \rangle$ 来表示语句 C 和对辅助变量 *aux* 的写操作是原子执行的，因此我们可以通过分析 *aux* 的值来跟踪程序的运行状态。然而。每个非原子的写操作在 TSO 中都必须先被缓存，因此为了让我们在验证某个线程代码的时候能够判断环境中其他线程写操作的状态（因为写缓存是局部的，验证中是看不到其他线程的写缓存的），因此我们需要通过辅助变量，追踪这些写操作是否处于缓存，或是已经实际写入内存。我们将辅助变量也看成是作用在普通内存地址上的写操作，$\langle [E_1] := E_1', [E_2] := E_2' \rangle$ 意味着内存写操作（第一个写）和辅助变量写操作 (第二个写) 应当原子地被同时放入缓存。并且在之后它们从缓存序列中出列并写入内存中时也是原子执行的。

语句 $\mathsf{CAS}(E_1, E_2, E_3)$ 是一个原子执行的 compare-and-swap 指令，这个指令可以看成是对 X86 中 *CMPXCHG* 指令的抽象。**mfence** 则是内存 fence 指令。在我们语言中，程序 *prog* 是由一个或者多个线程组成，而每个线程则是由线程 ID 和语句所组成。

5.1.1　TSO 标准模型

回忆我们在图 2.2中提到的 TSO 抽象机，现在我们可以在我们语言的基础上定义标准的 TSO 内存模型。TSO 的形式化机器模型在图 5.2中定义; 操作语义在图 5.3中定义，这些操作语义如 Owens 等人的工作 [19] 一样，同样也是标准化的定义，我们在此就不在赘述。

5.1.2　ATSO 内存模型

图 5.4中是 ATSO 模型的抽象表示。和冯新宇等人的 LRG [7] 的工作一样，我们在 “逻辑” 上将内存分为 n+1 块 (假设系统中有 n 个线程)：一个用来做共享内存，其他 n 块用于线程局部内存。抽象机模型在图 5.5中定义。状态 Σ 由共享内存和一组线程组成。每个线程拥有一个写缓存，一组寄存器以及一块局部内存。状态转移断言 R 和 G 是共享内存上的二元关系。

$$
\begin{array}{rrcl}
(TID) & tid & \in & Nat \qquad\qquad (Reg)\ r ::= r_1, r_2, \ldots \\
(Exp) & E & ::= & n \mid r \mid E_1 + E_2 \mid -E \mid \ldots \\
(BExp) & B & ::= & true \mid false \mid E_1 = E_2 \mid E_1 < E_2 \mid E_1 \neq E_2 \mid \ldots \\
(Instr) & \iota & ::= & [E] := E' \mid r := E \mid r := [E] \mid \langle [E_1] := E_1', [E_2] := E_2' \rangle \\
(Cmds) & \mathsf{C} & ::= & \iota \mid \textbf{mfence} \mid \mathsf{CAS}(E_1, E_2, E_3) \mid \textbf{skip} \\
 & & \mid & \textbf{if } B \textbf{ then } \mathsf{C}_1 \textbf{ else } \mathsf{C}_2 \mid \textbf{while } B \textbf{ do } \mathsf{C} \mid \mathsf{C}_1; \mathsf{C}_2 \\
(Prog) & prog & ::= & tid_1.\mathsf{C}_1 \parallel \cdots \parallel tid_n.\mathsf{C}_n
\end{array}
$$

图 5.1: 语言文法

$$
\begin{array}{rcl}
(Addr)\ a & \in & Nat \\
(Regfile)\ rf & \in & Reg \to Int \\
(Mem)\ m & \in & Addr \rightharpoonup_{fin} Int \\
(State)\ \sigma & ::= & (m, thds)
\end{array}
\qquad
\begin{array}{rcl}
(BufUpd) \quad u & ::= & (a, v) \mid ((a, v), (a', v')) \\
(Buffer)\ buf & ::= & [] \mid u :: buf \\
(Threads)\ thds & \in & TID \rightharpoonup_{fin} (Regfile \times Buffer)
\end{array}
$$

图 5.2: TSO 机器状态

和 LRG 一样，我们的操作语义也可以支持局部和共享内存之间动态的所有权转移。即在特定的指令执行后，可能会发生内存单元从共享内存转移至私有内存或者反之。为了描述转移前后共享内存的状态，这我们需要一个精确描述共享内存状态的 “precise”(节 5.2中的定义5.2.1) 的不变式 I 来指定在这些特定的指令执行以后，机器状态中的共享内存是什么样的。

在我们具体介绍操作语义之前，我们首先介绍一些在图 5.6中列出的重要的辅助定义。我们用 $\mathsf{dom}(buf)$ 来表示把写缓存 buf 中那些等待写入的写操作所对应的内存地址的集合，使用 $buf(a)$ 来表示写缓存 buf 中对地址 a 的 “最新” 的写操作的值——如同我们在之前介绍的那样，最新的意思即缓存中所有对 a 的写操作中, 最后放入缓存的那个。我们使用 $\blacktriangleright_I$ 和 $\triangleright_I$ 来分别表示将所有缓存的写按序写入内存中以及把缓存中最旧的写操作 (即在缓存中的队首元素) 写入内存。它们的形式化定义如在图 5.6所示。我们注意到，这两个符号都带有下标 I, 如前所述，当写操作从写缓存写入内存的时候，将会发生内存所有权转移，I 即用来指定转换前后的共享内存状态，用来指导抽象机的执行。

我们在图 5.7中给出 ATSO 的操作语义。我们在每条规则中都加入不变量 I 来描述在程序执行过程中，状态机所有可能的共享内存状态，用于指导那些能够对共享内存和局部内存之间所有权进行转移的命令进行所有权转移。图 5.7中的第一条规则，“state transition” 规则通过在线程局部状态和共享内存组成的元组上的转换规则来定义在全局机器状态上的转换规则。接下来两条规则分别定义从内存中读取单元值和从缓存中读取单元值。注意到我们的规则中要求共享

State Transition

$$\frac{\sigma = (m, thds) \quad thds(tid) = (rf, buf) \quad (\mathsf{C}, (m, rf, buf)) \rightsquigarrow (\mathsf{C}', (m', rf', buf'))}{(tid.\mathsf{C}, \sigma) \rightsquigarrow (tid.\mathsf{C}', (m', thds[tid \mapsto (rf', buf')]))}$$

Read from memory

$$\frac{[\![E]\!]_{rf} = a \quad m(a) = v \quad a \notin \mathsf{dom}(buf)}{(r := [E], (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m, rf[r \mapsto v], buf))}$$

Read from writer buffer

$$\frac{[\![E]\!]_{rf} = a \quad buf(a) = v}{(r := [E], (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m, rf[r \mapsto v], buf))}$$

Write to write buffer

$$\frac{[\![E]\!]_{rf} = a \quad [\![E']\!]_{rf} = v}{(([E] := E')_\alpha, (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m, rf, (a, v)_\alpha :: buf))}$$

Write from write buffer to memory

$$\frac{buf = buf' \texttt{++} [(a, v)_\alpha] \quad m' = m[a \mapsto v]}{(\mathsf{C}, (m, rf, buf)) \rightsquigarrow (\mathsf{C}, (m', rf, buf')}$$

Locked Instructions

$$\frac{(\iota, (m, rf, buf)) \rightsquigarrow^* (\mathbf{skip}, (m', rf', buf)) \quad buf = \mathbf{nil}}{(\mathbf{lock}\ \iota, (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m', rf', buf))}$$

Double Write to Write buffer

$$\frac{[\![E]\!]_{rf} = a \quad [\![E']\!]_{rf} = v \quad [\![E_1]\!]_{rf} = a' \quad [\![E_1']\!]_{rf} = v'}{(\langle [E] := E', [E_1] = E_1' \rangle_\alpha, (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m, rf, ((a, v), (a', v'))_\alpha :: buf))}$$

Double Write from write buffer to memory

$$\frac{buf = buf' \texttt{++} [((a, v), (a', v'))_\alpha] \quad m' = m[a \mapsto v, a' \mapsto v']}{(\mathsf{C}, (m, rf, buf)) \rightsquigarrow (\mathsf{C}, (m', rf, buf')}$$

CAS-FALSE

$$\frac{[\![E_1]\!]_{rf} = a \quad [\![E_2]\!]_{rf} = v \quad [\![E_3]\!]_{rf} = v' \quad m(a) \neq v \quad buf = \mathbf{nil}}{(r := \mathsf{CAS}(E_1, E_2, E_3)_\alpha, (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m, rf[r \mapsto 0], buf))}$$

CAS-TRUE

$$\frac{\begin{array}{c}[\![E_1]\!]_{rf} = a \quad [\![E_2]\!]_{rf} = v \quad [\![E_3]\!]_{rf} = v' \quad buf = \mathbf{nil} \\ m(a) = v \quad m' = m[a \mapsto v'] \quad rf' = rf[r \mapsto 1]\end{array}}{(r := \mathsf{CAS}(E_1, E_2, E_3)_\alpha, (m, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m', rf', buf))}$$

Program Step

$$\frac{(tid_i.\mathsf{C}_i, \sigma) \rightsquigarrow (tid_i.\mathsf{C}_i', \sigma')}{(tid_1.\mathsf{C}_1 \parallel \cdots \parallel tid_i.\mathsf{C}_i \parallel \ldots tid_n.\mathsf{C}_n, \sigma) \rightsquigarrow (tid_1.\mathsf{C}_1 \parallel \cdots \parallel tid_i.\mathsf{C}_i' \parallel \ldots tid_n.\mathsf{C}_n, \sigma')}$$

图 5.3: TSO 操作语义

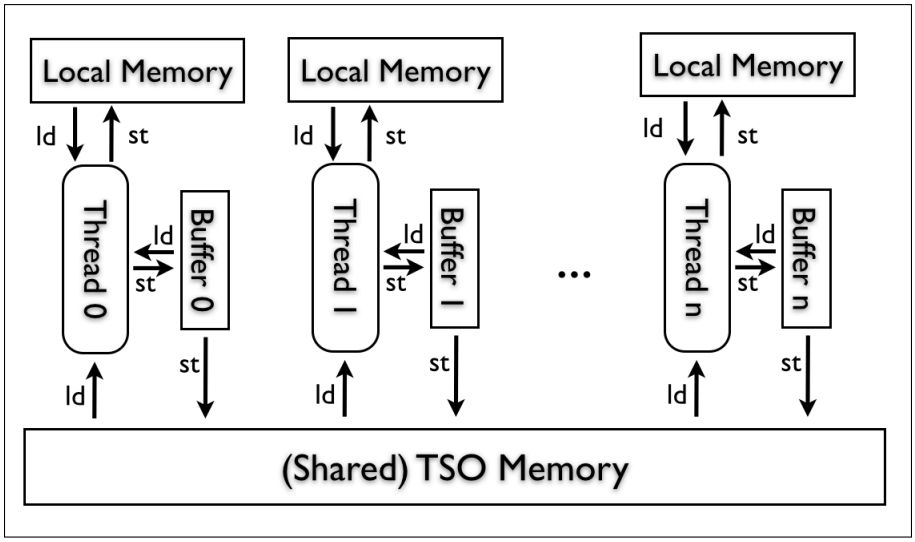


图 5.4: ATSO 抽象机

$$
\begin{array}{llll}
(Addr) & a & \in & Nat \\
(Value) & v & \in & Int \\
(Regfile) & rf & \in & Reg \to Int \\
(Mem) & m, m_l, m_s & \in & Addr \rightharpoonup_{fin} Int \\
(State) & \Sigma & ::= & (m, T)
\end{array}
\qquad
\begin{array}{llll}
(BufUpd) & u & ::= & (a,v) \mid ((a,v),(a',v')) \\
(Buffer) & buf & ::= & [] \mid u :: buf \\
(Threads) & T & \in & TID \rightharpoonup_{fin} \\
 & & & (Mem \times Regfile \times Buffer) \\
(Lstate) & \Delta & ::= & (m_s, m_l, rf, buf) \\
(Trans) & \mathcal{R}, \mathcal{G} & \in & \mathcal{P}(Mem \times Mem)
\end{array}
$$

图 5.5: ATSO 机器状态

内存部分需要满足不变式 I。我们使用 $\mathfrak{f}_1 \uplus \mathfrak{f}_2$ 来表示两个定义域不相交的部分函数的并集，使用 $\mathfrak{f}[\mathfrak{x} \mapsto \mathfrak{n}]$ 来表示更新函数 $\mathfrak{f}$ 在 $\mathfrak{x}$ 上的映射值为 $\mathfrak{n}$。而对表达式求值计算 $[\![\mathrm{E}]\!]_{rf}$ 以及 $[\![\mathrm{B}]\!]_{rf}$ 则是标准定义，我们使用存储在 rf 中的寄存器值来计算表达式的值：

$$
[\![B]\!]_{rf} \stackrel{\text{def}}{=} \begin{cases} n & \text{if } B = n \\ n & \text{if } B = r \wedge rf(r) = n \\ n_1 + n_2 & \text{if } B = B_1 + B_2 \wedge [\![B_1]\!]_{rf} = n_1 \wedge [\![B_2]\!]_{rf} = n_2 \end{cases}
$$

$$
[\![E]\!]_{rf} \stackrel{\text{def}}{=} \begin{cases} n & \text{if } E = n \\ n & \text{if } E = r \wedge rf(r) = n \\ n_1 + n_2 & \text{if } E = E_1 + E_2 \wedge [\![E_1]\!]_{rf} = n_1 \wedge [\![E_2]\!]_{rf} = n_2 \end{cases}
$$

对于写操作，我们将对局部内存和共享内存的写操作分开来，对局部内存的写操作是立即发生的，而对共享内存的写操作将先被放入缓存当中。同时，从写缓存中取出并对共享内存进行写操作会导致共享内存和局部内存对内存单元所有权的转移。

$$\mathsf{dom}(buf) \stackrel{\text{def}}{=} \begin{cases} \emptyset & buf = \mathbf{nil} \\ \{a\} \cup \mathsf{dom}(buf') & buf = (a, \_)_\alpha :: buf' \\ \{a, a'\} \cup \mathsf{dom}(buf') & buf = ((a, \_), (a', \_))_\alpha :: buf' \end{cases}$$

$$buf(a) \stackrel{\text{def}}{=} \begin{cases} v & buf = (a, v)_\alpha :: buf' \text{ or } buf = ((a, v), (a', v'))_\alpha :: buf' \\ buf'(a) & buf = (a', \_)_\alpha :: buf' \text{ or } buf = ((a', \_), (a'', v''))_\alpha :: buf', \text{ and } a' \neq a \\ undef & buf = \mathbf{nil} \end{cases}$$

$$\frac{a \subseteq \mathsf{dom}(m_s \uplus m_l) \quad (m_s' \uplus m_l') = (m_s \uplus m_l)[a \mapsto v] \quad (m_s, rf) \models \mathrm{I}}{(m_s, m_l, rf, buf{+\!+}[(a, v)]) \rhd_\mathrm{I} (m_s', m_l', rf, buf)}$$

$$\frac{\{a, a'\} \subseteq \mathsf{dom}(m_s \uplus m_l) \quad (m_s' \uplus m_l') = (m_s \uplus m_l)[a \mapsto v, a' \mapsto v'] \quad (\{m_s, m_s'\}, rf) \models \mathrm{I}}{(m_s, m_l, rf, buf{+\!+}[((a, v), (a', v'))]) \rhd_\mathrm{I} (m_s', m_l', rf, buf)}$$

$$\frac{}{\Delta \rhd_\mathrm{I}^* \Delta} \qquad \frac{\Delta \rhd_\mathrm{I} \Delta'' \quad \Delta'' \rhd_\mathrm{I}^* \Delta'}{\Delta \rhd_\mathrm{I}^* \Delta'}$$

$$\blacktriangleright_\mathrm{I} \Delta \stackrel{\text{def}}{=} \begin{cases} \Delta' & \text{if } \Delta \rhd_\mathrm{I}^* \Delta' \text{ and } \Delta'.buf = \mathbf{nil} \\ undef & \text{if there is no such } \Delta' \end{cases}$$

图 5.6: ATSO 辅助定义

我们观察到这么一点，在没有锁或者其他同步操作的情况下，把局部内存单元转移到共享内存中，是安全的 (因为局部内存之前对其他线程来说不可见)，反过来，将某块原本属于共享内存的单元变成局部内存所有则是不安全的。因为如果我们允许某个线程的非原子写操作引起的内存所有权转移能从共享资源转移到局部的话，那么由于这个线程看到这个非原子写操作的时间要早于其他线程（因为非原子写操作先被放入写缓存中），那么这个线程对共享资源的观察可能就会迥异与其他线程，这样的不一致性会导致数据竞争。因此我们规定，在对共享内存进行写操作时候所能进行的所有权转移，只能从局部内存到共享内存。

因此，当我们在对某块内存单元进行写操作的时候，如果该内存单元已经是共享内存，那么我们只需要把这个写操作放入写缓存中。但是如果该内存单元是局部内存，那么我们就需要去分析，当机器状态的写缓存中所有的写操作都写入内存时，该内存单元所有权的归属。如果该内存单元仍然是局部单元，说明在这一过程中，该内存单元的所有权一直属于局部内存，因此我们可以直接对该局部内存单元进行写操作而不用缓存；如果该内存单元在清空写缓存 (这边清空的意思是指将缓存中的所有写操作按序写入内存中，下同) 以后属于共享内存单元，那么我们对该内存单元的写操作就要视为是对共享内存单元的写操作，需要放入写缓存当中。

State Transition

$$\frac{\Sigma = (m_s, T) \quad T(tid) = (m_l, rf, buf) \quad \mathrm{I} \vdash (\mathrm{C}, (m_s, m_l, rf, buf)) \rightsquigarrow (\mathrm{C}', (m_s', m_l', rf', buf'))}{\mathrm{I} \vdash (tid.\mathrm{C}, \Sigma) \rightsquigarrow (tid.\mathrm{C}', (m_s', T[tid \mapsto (m_l', rf', buf')]))}$$

Read from memory

$$\frac{[\![E]\!]_{rf} = a \quad (m_s \uplus m_l)(a) = v \quad a \notin \mathsf{dom}(buf) \quad (m_s, rf, buf) \models \mathrm{I}}{\mathrm{I} \vdash (r := [E], (m_s, m_l, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m_s, m_l, rf[r \mapsto v], buf))}$$

Read from write buffer

$$\frac{[\![E]\!]_{rf} = a \quad buf(a) = v \quad (m_s, rf, buf) \models \mathrm{I}}{\mathrm{I} \vdash (r := [E], (m_s, m_l, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m_s, m_l, rf[r \mapsto v], buf))}$$

Write to write buffer

$$\frac{[\![E]\!]_{rf} = a \quad [\![E']\!]_{rf} = v \quad \blacktriangleright_{\mathrm{I}} (m_s, m_l, rf, buf) = (m_s', m_l', rf, \mathbf{nil}) \quad a \in \mathsf{dom}(m_s')}{\mathrm{I} \vdash ([E] := E', (m_s, m_l, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m_s, m_l, rf, (a, v) :: buf))}$$

Direct Write to memory

$$\frac{[\![E]\!]_{rf} = a \quad [\![E']\!]_{rf} = v \quad \blacktriangleright_{\mathrm{I}} (m_s, m_l, rf, buf) = (m_s', m_l', rf, \mathbf{nil}) \quad a \notin \mathsf{dom}(m_s') \quad a \in \mathsf{dom}(m_l)}{\mathrm{I} \vdash ([E] := E', (m_s, m_l, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m_s, m_l[a \mapsto v], rf, buf))}$$

Double Write to write buffer

$$\frac{\begin{array}{c}[\![E_1]\!]_{rf} = a_1 \quad [\![E_2]\!]_{rf} = v_1 \quad [\![E_3]\!]_{rf} = a_2 \quad [\![E_4]\!]_{rf} = v_2 \\ \blacktriangleright_{\mathrm{I}} (m_s, m_l, rf, buf) = (m_s', m_l', rf, \mathbf{nil}) \quad a_1, a_2 \in \mathsf{dom}(m_s')\end{array}}{\mathrm{I} \vdash (\langle [E_1] := E_2, [E_3] := E_4 \rangle, (m_s, m_l, rf, buf)) \rightsquigarrow (\mathbf{skip}, (m_s, m_l, rf, ((a_1, v_1), (a_2, v_2)) :: buf))}$$

Seq Compose

$$\frac{\mathrm{I} \vdash (\mathrm{C}_1, \Delta) \rightsquigarrow (\mathbf{skip}, \Delta')}{\mathrm{I} \vdash (\mathrm{C}_1; \mathrm{C}_2, \Delta) \rightsquigarrow (\mathrm{C}_2, \Delta')}$$

Write from write buffer to memory with ownership transfer

$$\frac{\Delta \rhd_{\mathrm{I}} \Delta'}{\mathrm{I} \vdash (\mathrm{C}, \Delta) \rightsquigarrow (\mathrm{C}, \Delta')}$$

CAS-TRUE with ownership transfer

$$\frac{\begin{array}{c}[\![\mathsf{E}_1]\!]_{rf} = \mathfrak{a} \quad [\![\mathsf{E}_2]\!]_{rf} = \mathfrak{v} \quad [\![\mathsf{E}_3]\!]_{rf} = \mathfrak{v}' \quad m_s(\mathfrak{a}) = \mathfrak{v} \\ (m_s' \uplus m_l') = (m_s \uplus m_l)[a \mapsto \mathfrak{v}'] \quad rf' = rf[\mathsf{r} \mapsto 1] \quad (m_s', rf', buf) \models \mathrm{I} \quad (m_s, rf, buf) \models \mathrm{I}\end{array}}{\mathrm{I} \vdash (\mathsf{r} := \mathsf{CAS}(\mathsf{E}_1, \mathsf{E}_2, \mathsf{E}_3), (m_s, m_l, rf, \mathbf{nil})) \rightsquigarrow (\mathbf{skip}, (m_s', m_l', rf', \mathbf{nil}))}$$

CAS-FALSE

$$\frac{\begin{array}{c}[\![\mathsf{E}_1]\!]_{rf} = \mathfrak{a} \quad [\![\mathsf{E}_2]\!]_{rf} = \mathfrak{v} \quad m_s(\mathfrak{a}) \neq \mathfrak{v} \\ rf' = rf[\mathsf{r} \mapsto 0] \quad (m_s, rf, buf) \models \mathrm{I}\end{array}}{\mathrm{I} \vdash (\mathsf{r} := \mathsf{CAS}(\mathsf{E}_1, \mathsf{E}_2, \mathsf{E}_3), (m_s, m_l, rf, \mathbf{nil})) \rightsquigarrow (\mathbf{skip}, (m_s', m_l', rf', \mathbf{nil}))}$$

Program Step

$$\frac{\mathrm{I} \vdash (tid_{\mathrm{i}}.\mathrm{C}_{\mathrm{i}}, \Sigma) \rightsquigarrow (tid_{\mathrm{i}}.\mathrm{C}_{\mathrm{i}}', \Sigma')}{\mathrm{I} \vdash (tid_1.\mathrm{C}_1 \parallel \cdots \parallel tid_{\mathrm{i}}.\mathrm{C}_{\mathrm{i}} \parallel \ldots tid_{\mathrm{n}}.\mathrm{C}_{\mathrm{n}}, \Sigma) \rightsquigarrow (tid_1.\mathrm{C}_1 \parallel \cdots \parallel tid_{\mathrm{i}}.\mathrm{C}_{\mathrm{i}}' \parallel \ldots tid_{\mathrm{n}}.\mathrm{C}_{\mathrm{n}}, \Sigma')}$$

图 5.7: ATSO 操作语义

图 5.7中的这条规则 “Write to write buffer” 说的就是是如果当我们把缓存都清空后，目标地址 a 属于共享内存，那么我们将该写操作放入写缓存中。注意 $\blacktriangleright_{\mathrm{I}}$ 的在图 5.6中的定义，我们使用 $\blacktriangleright_{\mathrm{I}}\Delta$ 来表示清空 Δ 中的缓存。注意在这里 m_s 的定义域可能比 m_s' 的定义域小，因为如上面所述，在清空缓存的过程中，可能会导致局部内存单元变成共享内存单元，因此一个对地址 a 的写操作是局部写操作只有当清空缓存以后 a 仍然属于局部内存才成立。下一条规则 “Direct Write to memory” 说的是如果在缓存清空以后，目标地址属于局部内存，那么该写操作可以直接作用在内存地址上而不需要缓存。

接下来这条规则 “Double Write to write buffer” 和之前的 “write to write buffer” 非常类似，唯一的区别在于我们需要原子地把对内存单元和辅助变量的写操作同时放入写缓存当中。

如同规则 “Write from write buffer to memory with ownership transfer” 规则描述的那样，被缓存的写操作可以非确定性的从缓存中出队并执行。我们使用 $\triangleright_{\mathrm{I}}$ 代表将缓存中的队首写操作 (即缓存中目前最早被放入的写操作) 出队并写入内存，它的形式化定义如图 5.6中所示。如同我们上面提到的那样，当一个缓存写操作被执行，我们除了将该写操作写入内存外，我们同时还能调整共享内存和局部内存之间的边界，即可以将某些局部内存单元转换为共享内存单元，而如何转换则是根据不变量 I 的规范所决定。

接下来我们可以看 CAS 语句是如何执行的，这体现在 “CAS-TRUE” 和 “CAS-FALSE” 这两条规则中。我们注意到规则中要求，CAS 语句只能在缓存为空的状态下执行。当 CAS 判断条件为真的时候，“CAS-TRUE” 可以直接往内存中进行写操作 (无论所写的地址是共享或是局部)，同样还能根据不变量 I 来调整共享内存和局部内存之间的边界 (和写缓存的内存所有权转移只能从局部转移至共享内存不同，CAS 的转移可以是双向的，因为 CAS 指令可以视为对所有内存加上一个同步锁，因此不管是从局部到共享还是从共享到局部，都是线程安全的)。当判断条件为假的时候，我们使用 “CAS-FALSE” 规则，只是简单地将寄存器置的值置为 0。

图 5.7中的最后一条规则用于并发环境的操作，是整个内存模型操作语义的顶级规则。

5.2　断言定义

我们的断言语言设计如图 5.8所示。我们将断言分为三级，其中第一级 P, Q, I 是标准的分离逻辑断言，只用来描述内存状态和寄存器状态。第二级 tp, tq 用来描述单个线程以及共享内存的状态 $tp, \mathtt{tq}, \mathrm{I}$。第三级是顶级断言，用来描述整个机器状态 q, p。状态转移动作 R，G 和 A 用来描述在共享内存上的状态转移，这可以用来定义 rely 和 guarantee 条件。

$$
\begin{array}{rrcl}
(\mathit{BuffAst}) & \beta & ::= & E_1 \rightharpoonup E_2 \mid (E_1, E_2) \rightharpoonup (E_1', E_2') \\
(\mathit{SepAst}) & P, Q, \mathrm{I} & ::= & \mathbf{emp} \mid E \mapsto E' \mid B \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \exists \mathsf{X}.P \mid \blacktriangleright tp \\
(\mathit{ThdAst}) & tp, tq & ::= & P \mid \beta \triangleright tp \mid E \hookrightarrow E' \mid \lfloor tp \rfloor \mid tp * tq \mid \ldots \\
(\mathit{StateAst}) & p, q & ::= & \{\!|tp|\!\}^{tid} \mid p \wedge q \mid p \vee q \mid p * q \\
(\mathit{Action}) & R, G, A & ::= & P \ltimes Q \mid [P] \mid A \vee A \mid \exists \mathsf{X}.A
\end{array}
$$

图 5.8: 断言语言

$$
m_1 \uplus m_2 \stackrel{\text{def}}{=} \begin{cases} m_1 \cup m_2 & \text{if } \mathsf{dom}(m_1) \cap \mathsf{dom}(m_2) = \emptyset \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
buf_1 \sharp buf_2 \stackrel{\text{def}}{=} \begin{cases} \{buf_1 \texttt{++} buf_2\} & \text{if } buf_1 = \mathbf{nil} \text{ or } buf_2 = \mathbf{nil} \\ \{u :: buf' \mid buf_1 = u :: buf_1', buf' \in (buf_1' \sharp buf_2)\} & \\ \quad \cup \{u :: buf' \mid buf_2 = u :: buf_2', buf' \in (buf_1 \sharp buf_2')\} & \text{otherwise} \end{cases}
$$

$$
buf_1 \uplus buf_2 \stackrel{\text{def}}{=} \begin{cases} buf_1 \sharp buf_2 & \text{if } \mathsf{dom}(buf_1) \cap \mathsf{dom}(buf_2) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}
$$

$$
\Delta_1 \uplus \Delta_2 \stackrel{\text{def}}{=} \{(\Delta_1.m_s \uplus \Delta_2.m_s, \Delta_1.m_l \uplus \Delta_2.m_l, rf, buf) \mid buf \in (\Delta_1.buf \uplus \Delta_2.buf), rf = \Delta_1.rf = \Delta_2.rf\}
$$

$$
\begin{array}{ll}
(m_s, m_l, rf, buf) \models_{\mathrm{I}} B & \mathit{iff}\ [\![B]\!]_{rf} = \mathit{true} \\
(m_s, m_l, rf, buf) \models_{\mathrm{I}} \mathbf{emp} & \mathit{iff}\ m = \emptyset \wedge buf = \mathbf{nil} \wedge (m_s, rf) \models_{SL} \mathrm{I} \\
(m_s, m_l, rf, buf) \models_{\mathrm{I}} E_1 \mapsto E_2 & \mathit{iff}\ \exists a, v.\, \mathsf{dom}(m_s \uplus m_l) = \{a\} \wedge (m_s \uplus m_l)(a) = v \\
 & \quad \wedge buf = \mathbf{nil} \wedge (m_s, rf) \models_{SL} \mathrm{I} \\
\Delta \models_{\mathrm{I}} \lfloor tp \rfloor & \mathit{iff}\ \exists rf, buf.\, (\Delta.m_s, \Delta.m_l, rf, buf) \models tp \\
\Delta \models_{\mathrm{I}} tp * tq & \mathit{iff}\ \exists \Delta_1, \Delta_2.\, \Delta \in \Delta_1 \uplus \Delta_2 \wedge (\Delta_1 \models_{\mathrm{I}_1} tp) \wedge (\Delta_2 \models_{\mathrm{I}_2} tq) \\
 & \quad \wedge \mathrm{I} = \mathrm{I}_1 * \mathrm{I}_2 \\
\Delta \models_{\mathrm{I}} \blacktriangleright tp & \mathit{iff}\ \delta.buf = \mathbf{nil} \wedge \exists \Delta'.\, (\Delta' \models_{\mathrm{I}} tp) \wedge (\Delta' \rhd_{\mathrm{I}}^{*} \Delta)
\end{array}
$$

$$
\begin{array}{l}
(m_s, m_l, rf, buf) \models_{\mathrm{I}} (E_1 \rightharpoonup E_2) \triangleright tp \ \mathit{iff} \\
\qquad \exists a, v.\, [\![E_1]\!]_{rf} = a \wedge [\![E_2]\!]_{rf} = v \wedge a \in \mathsf{dom}(m_l \uplus m_s) \\
\qquad \wedge ((\exists buf'.\, buf = (a, v) :: buf' \wedge (m_s, m_l, rf, buf') \models_{\mathrm{I}} tp) \\
\qquad\quad \vee (buf = \mathbf{nil} \wedge m_s(a) = v \wedge \exists v'.\, (m_s[a \mapsto v'], m_l, rf, \mathbf{nil}) \models_{\mathrm{I}} \blacktriangleright tp))
\end{array}
$$

$$
\begin{array}{ll}
(m_s, m_l, rf, buf) \models_{\mathrm{I}} E_1 \hookrightarrow E_2 & \mathit{iff}\ \exists a, v.\, [\![E_1]\!]_{rf} = a \wedge [\![E_2]\!]_{rf} = v \\
 & \quad \wedge (buf(a) = v \vee a \notin \mathsf{dom}(buf) \wedge (m_s \uplus m_l)(a) = v)
\end{array}
$$

$$
\begin{array}{ll}
\Sigma \models_{\mathrm{I}} tp^{tid} & \mathit{iff}\ (\Sigma.m, m_l, rf, buf) \models_{\mathrm{I}} tp \wedge \Sigma.T(tid) = (m_l, rf, buf) \wedge \mathsf{dom}(\Sigma.T) = \{tid\} \\
\Sigma \models_{\mathrm{I}} p \wedge q & \mathit{iff}\ \Sigma \models_{\mathrm{I}} p \wedge \Sigma \models_{\mathrm{I}} q \\
\Sigma \models_{\mathrm{I}} p \vee q & \mathit{iff}\ \Sigma \models_{\mathrm{I}} p \vee \Sigma \models_{\mathrm{I}} q \\
(m, T) \models_{\mathrm{I}} p * q & \mathit{iff}\ \exists T_1, T_2.\, (m, T_1) \models_{\mathrm{I}} p \wedge (m, T_2) \models_{\mathrm{I}} q \wedge T = T_1 \uplus T_2
\end{array}
$$

图 5.9: 断言语义

$$
\begin{array}{ll}
((m_s, m_l, rf, buf), (m_s', m_l', rf', buf')) \models P \ltimes Q & \textit{iff } (m_s, rf) \models_{\mathsf{SL}} P \wedge (m_s', rf') \models_{\mathsf{SL}} Q \\
((m_s, m_l, rf, buf), (m_s, m_l, rf, buf)) \models [\mathsf{P}] & \textit{iff } (m_s, rf) \models_{\mathsf{SL}} P \\
(\Delta, \Delta') \models A \vee A' & \textit{iff } (\Delta, \Delta') \models A \textit{ or } (\Delta, \Delta') \models A' \\
A \Longrightarrow A' & \stackrel{\text{def}}{=} \text{for all } \Delta \text{ and } \Delta', \text{ if } (\Delta, \Delta') \models A, \\
 & \qquad \text{then } (\Delta, \Delta') \models A' \\
[\![A]\!] & \stackrel{\text{def}}{=} \{(\Delta, \Delta') \mid (\Delta, \Delta') \models A\} \\
\mathsf{True} & \stackrel{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true} \\
\mathsf{Id} & \stackrel{\text{def}}{=} [\mathsf{true}]
\end{array}
$$

图 5.10: 状态变换动作语义

设计 TSO 断言的难点在于，我们需要设计一个稳定的断言。当我们的断言中带有缓存描述以后，我们能够知道满足该断言的机器状态所对应的缓存应该是什么样。但是，由于缓存的存在，这个断言所描述机器状态并不是 “稳定” 的, 这里的不稳定是指，这个机器状态在任意时刻都能够通过执行缓存中的写操作从而到达另外一个机器状态。因为当我们使用断言来进行程序验证条件推导的时候，当某个线程在执行往某条语句之后，在 SC 模型下并且不考虑其他线程交互的情况下，机器状态是稳定的，我们很容易可以用断言表达出来。然而，在 TSO 模型下，当某个线程执行语句之后，由于缓存的存在，除非缓存为空，否则其机器状态是不稳定的，随时可以通过执行缓存中的写操作来到达另一个状态。

举个例子, 假设我们有 Hoare 风格的三元组 {P}x := 1{Q}。这个三元组的含义是指，当机器状态满足 P 的时候，成功执行完 x := 1 这条语句后，机器状态应当满足 Q。我们知道，在 TSO 模型下，执行 x := 1 这条语句会先把这个写操作放入缓存中。因此 Q 说的是写缓存中有若干写操作，并且最后一个写操作是对 x 写入 1。但是，这样的断言 Q 是不稳定的，因为那些在写缓存中的写操作在这条赋值语句做完以后的任何时刻都可能会被从缓存队列中取出并写入内存，那么所到达的状态我们知道就不再满足 Q 了。

所以，我们的断言如果仅仅是做到描述某一个机器状态中它的缓存是长什么样的是远远不够的，我们需要要求我们的断言能够满足这些所有的通过执行缓存写而到达的状态，即我们的断言关于写缓存是稳定的，否则，我们的逻辑就没有实际运用的意义。这也是我们接下来要设计断言语义时的思路，我们会通过巧妙的设计，让我们通过断言语言所所写出来的任何断言都是关于缓存稳定的。这也是我们这个工作的另一个贡献点。关于 “缓存稳定” 的形式化定义，请参见定义5.2.2, 我们同时还会给出我们的断言是关于缓存稳定的引理和相关证明，参见引理5.2.1。

我们把断言语义放在图 5.9中。接下来我们将会一个一个地详细介绍这些语义。首先，我们使用 $\uplus$ 这个二元关系符来表示其左右两个集合的不相交并集，当且仅当两者不相交时才有定义。我们在这边引入缓存间的不相交并集的概念。

我们知道，在 TSO 模型下，缓存可以视为先进先出队列，在这个队列中的所有元素都有一个全序关系。而两个不同的缓存之间的元素则没有全序关系。因此我们使用 $buf_1 \sharp buf_2$ 表示将两个缓存合并成一个新的缓存的集合，而对于该集合中的任意元素之间，都要保持我们上面所说的关系，即如果两个元素来自于合并前的同一个缓存，比如 buf_1, 则它们在任意属于合并缓存集合的缓存中存在与在 buf_1 中一样的先后关系。形式化定义参考图 5.9中 $buf_1 \sharp buf_2$ 中的定义。而如果两个合并的缓存之间，它们的定义域 $\mathsf{dom}(buf)$(缓存定义域的概念请参考图 5.6中的定义) 不相交，则我们就能有 $buf_1 \uplus buf_2$, 这就是 buf_1 和 buf_2 的不相交并集。有了这个概念以后，我们就能定义 $\Delta_1 \uplus \Delta_2$。由此，我们就能够像分离逻辑那样定义出在线程断言 tp 和 tq 之间 Separation Conjunction 符号 “*” 的语义。

$(m_s, m_l, rf, buf) \models_{\mathrm{I}} B$ 的语义很简单，即要求 B 通过寄存器文件 rf 求值为真即可，由于缓存写而改变的机器状态转移并不会改变寄存器的值，所以这条断言显然是缓存稳定的。$(m_s, m_l, rf, buf) \models_{\mathrm{I}} \mathbf{emp}$ 和 $(m_s, m_l, rf, buf) \models_{\mathrm{I}} E_1 \mapsto E_2$ 在分离逻辑的基础上，还额外要求了满足断言的机器状态缓存为空，并且共享内存要满足不变量 I（我们用 $(m_s, rf) \models_{\mathsf{SL}} \mathrm{I}$ 表示共享内存需要在分离逻辑的语义下满足断言 I)。由于这两条断言要求缓存为空，因此也是缓存稳定的。

$E_1 \hookrightarrow E_2$ 的语义也很简单，它的意思是说，满足该断言的四元组机器状态中，要么缓存中最新的对地址 $[\![E_1]\!]$ 的写操作的值是 $[\![E_2]\!]$，要么缓存中没有对该地址的写，并且内存中地址 $[\![E_1]\!]$ 的值是 $[\![E_2]\!]$。这个断言显然也是缓存稳定的。

缓存断言。接下来要介绍的两个断言语义是我们的断言逻辑中最难理解的部分，我们在介绍完之后会引入例子来帮组读者更好地理解。首先是 $\Delta \models_{\mathrm{I}} \blacktriangleright tp$。这个断言语义说的是，满足 $\blacktriangleright tp$ 的四元组 Δ 符合两个条件：第一，Δ 的缓存为空；第二，存在这么一个 Δ', 使得 Δ' 满足 tp, 并且 Δ 是 Δ' 清空其缓存后所得。由于 Δ 的缓存为空，所以这个断言是缓存稳定的。

其次是 $(m_s, m_l, rf, buf) \models_{\mathrm{I}} (E_1 \rightharpoonup E_2) \triangleright tp$. 这个断言语义的直观含义是，如果有一个四元组 (m_s, m_l, rf, buf') 满足 tp, 那么，当我们往 buf' 的队尾加入 $([\![E_1]\!], [\![E_2]\!])$ 的写操作后所得的新的四元组将满足 $(E_1 \rightharpoonup E_2) \triangleright tp$, 并且，从这个新的四元组状态出发清空缓存所能到达的所有状态 (我们使用 $\delta \rhd_{\mathrm{I}} \delta'$, 具体定义参见图 5.6表达)，也都满足 $(E_1 \rightharpoonup E_2) \triangleright tp$。同样，这个断言也是缓存稳定的。

上面所述的是我们断言中的第一级线程断言的语义。图 5.9中接下来的几条是第二级机器状态断言的语义。我们在机器状态断言上会加上一个上标 tid 用来表示线程 ID, 以区分不同线程间的局部内存和寄存器。在机器状态断言上的 Separation conjunction $*$ 符号则和 Viktor 在 [8] 中一样，对于寄存器，缓存，局部内存来说是 “multiplicative” 的而对共享内存来说则是 “additive” 的。

例 5.2.1. 接下来两个例子将会展示我们的断言是如何表达的：

- 假设我们有 $tp \stackrel{\text{def}}{=} (a_2 \rightharpoonup 200) \triangleright (a_1 \rightharpoonup 100) \triangleright (a_1 \mapsto 0 * a_2 \mapsto 0 * a_3 \mapsto 0)$, 以及

$\mathrm{I} = (a_1 \mapsto \_ * a_2 \mapsto \_)$ 不变式 I 作用在断言 *tp* 上，根据断言语义，我们知道机器状态局部内存中只有地址 a_3 一个单元，而共享内存上有 a_1, a_2 两个单元。它们保存的值分别为 *0*。同时在缓存中有对地址 a_1 和 a_2 分别写入 *100* 和 *200* 的缓存写。并且，从这个机器状态开始通过清空写缓存所能到达的任何状态也都满足 *tp*。

- 我们让 $tq \stackrel{\text{def}}{=} (a_1 \rightharpoonup 500) \triangleright tp$，*tp* 如上面所定义并且有 $\mathrm{I}' = \mathrm{I} \vee (a_1 \mapsto 500 * a_2 \mapsto \_ * a_3 \mapsto \_)$. 我们注意到，$\mathrm{I}'$ 作用在 *tq* 上的语义，意味着当对 a_1 写入 *500* 时，会发生一个从局部内存到共享内存的所有权转移，即把 a_3 从局部内存转移至共享内存。

Actions 的语义 (rely/guarantee 条件) 定义在图 5.10中。状态转移动作 $tp \ltimes tq$ 表示从初始共享内存满足 *tp* 到结果共享内存满足 *tq* 的状态转换。状态转移动作 [*tp*] 表示转移前后状态不变，并且满足 *tp*。我们注意到，我们这边使用 $\models_{\mathsf{SL}}$ 来定义共享内存满足断言，这说明我们在 Actions 中所写的断言形式只能有最基本的分离逻辑的部分，而不包含我们新添的描述缓存的部分，因为我们在这边只需要关心共享内存的变化即可：Rely 部分描述的是环境对共享内存的影响，Guarantee 部分描述的是线程自身对共享内存的改变。而描述缓存部分则是线程局部的资源，因此不应该出现在 rely/guarantee 条件中。

接下来，我们将给出 “精确断言”（定义5.2.1）, 缓存稳定 (定义5.2.2) 以及环境稳定的形式化定义 (定义5.2.3):

定义 5.2.1 (精确断言)**.** 我们说断言 I 是精确的，即 $\mathsf{precise}(\mathrm{I})$ 成立，当且仅当对于任意的 $m_s, m_{s1}, m_{s2}, rf,$，若 $m_{s1} \subseteq m_s, m_{s2} \subseteq m_s, (m_{s1}, rf) \models_{\mathsf{SL}} \mathrm{I}, (m_{s2}, rf) \models_{\mathsf{SL}} \mathrm{I},$ 则有 $m_{s1} = m_{s2}$。

定义 5.2.2 (缓存稳定)**.** 我们说 *tp* 是缓存稳定的，当且仅当, 对于任意的 Δ, Δ' 使得 $\Delta \models_{\mathrm{I}} tp$ 以及 $\Delta \triangleright_{\mathrm{I}}^{*} \Delta',$ 都有 $\Delta' \models_{\mathrm{I}} tp$。

引理 5.2.1. 所有的线程断言 *tp* 和 *tq* 都是缓存稳定的。

证明. 我们可以通过对断言 *tp* 和 *tq* 进行归纳，很容易得出结果. □

推论 5.2.1. 对于 Σ 和 *tid,* 如果 $\Sigma \models_{\mathrm{I}} tp^{\mathrm{t}}$, 那么我们有对于任意的 C 以及 Σ', 若 $(tid.\mathrm{C}, \Sigma) \rightsquigarrow^{*} (tid.\mathrm{C}, \Sigma')$, 则 $\Sigma' \models_{\mathrm{I}} tp^{\mathrm{t}}$。

根据我们前面提到的 actions 的语义，我们可以引入线程断言 *tp* 在环境 *R* 下稳定的概念：

定义 5.2.3 (环境稳定)**.** 我们说 *tp* 是关于 *R* 稳定的, 即 $\mathsf{sta}(tp, \mathrm{R})$ 成立, 当且仅当对于任意的 m_s, m_s', m_l, buf, rf I,
若:

$$(m_{s1}, m_l, rf, buf) \models_{\mathrm{I}} tp \wedge (m_{s1}, m_{s2}) \models \mathrm{R}$$

则有:

$$(m_{s2}, m_l, rf, buf') \models_{\mathrm{I}} tp$$

跟冯新宇的工作 [7] 一样，我们还需要引入 “Invariant-Fenced Actions” 的概念, 用来建立不变量 I 和 rely/guarantee 条件之间的对应关系。

定义 5.2.4 (Fence). $\mathrm{I}\diamond A$ *成立当且仅当* $[\mathrm{I}] \Longrightarrow A, A \Longrightarrow (\mathrm{I} \ltimes \mathrm{I})$ *以及* $\mathsf{precise}(\mathrm{I})$。

引理 5.2.2. *对于任意的* $\mathrm{I}, A, tp, tq,$ *若* $\mathrm{I}\diamond A$ *并且* $(tp \ltimes tq) \Longrightarrow A,$ *那么我们有* $(tp \vee tq) \Longrightarrow \mathrm{I}$。

证明. 通过展开 Def. 5.2.4, 我们有 $[\mathrm{I}] \Longrightarrow A$ 以及 $A \Longrightarrow (\mathrm{I} \ltimes \mathrm{I})$。因此我们知道 $(tp \ltimes tq) \Longrightarrow (\mathrm{I} \ltimes \mathrm{I})$。于是我们有 $tp \Longrightarrow \mathrm{I}$ 和 $tq \Longrightarrow \mathrm{I}$，所以可以有 $(tp \vee tq) \Longrightarrow \mathrm{I}$。 □

5.3　逻辑系统

图 5.11中是我们证明系统的推理规则。

与 LRG 类似的，我们使用 judgment 这么一个五元组来定义在线程执行 C 的过程中，机器状态是如何被改变的。Judgment 形如 $R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\}$. 其中 R 和 G 分别是依赖/保证条件。而它们都被描述共享内存的不变量 I 所 “fence”（见定义5.2.4)。tp 和 tq 分别是前后条件。值得注意的是 R, G, I 都只描述了共享内存，而 tp 和 tq 则描述了共享内存和线程局部的内存与缓存。

L-ASSN 规则的意思是，如果没有任何的共享资源 (R, G 是 **Emp**, 而 I 是 **emp** 描述了这一点)，那么对 E_1 的写操作就如同普通分离逻辑那样，直接写入内存，即把后断言的 $[E_1]$ 的值置为 E_2。

下一条规则用于推导共享内存的写操作。首先我们注意到此时环境依赖条件是 [I], 说明此时线程所处的环境不会对共享状态产生任何影响。这条规则有两个前提，第一个前提说的是，当把当前线程的缓存清空以后，E_1 处在共享内存当中。注意到我们在这边使用使用了 P 和 Q 这两个内存断言，根据我们上面一节的定义，这两个断言是标准的分离逻辑断言，只描述了内存状态。在这条前提中，它们的语义分别是：Q 代表在当前线程缓存清空以后，仍然存在于局部内存中部分；P 代表的是，在进行对 E_1 的写操作执行并且同时发生的内存所有权转移后，共享内存中除了 E_1 以外的部分。其中隐含了将 Q 所代表的局部内存转移到了 P 所代表的共享内存中。当然，当 Q 为 **emp** 时则表示没有发生内存所有权转移。在这种情况下，我们的后条件直接用内存断言 $\triangleright$ 将写操作和前条件连接在一起来表示将这个写操作放入缓存中。

S-ASSN-D 是用于共享内存的双写操作的推导。这条规则和上面一条基本类似，除了双写是同时把两个写操作放入写缓存，并且也是同时从写缓存中取出写入内存的。

ATOM-R 规则负责把上面两条环境不会对共享内存进行改变时能够使用的规则应用到环境会对共享内存产生影响的情形下。此时，我们需要额外要求，前后条件在环境的影响下是稳定的，并且 I 能够 fence 依赖/保证条件 R/G。

$$\frac{}{\mathbf{Emp}, \mathbf{Emp}, \mathbf{emp} \vdash \{E_1 \mapsto \_\}[E_1] = E_2\{E_1 \mapsto E_2\}} \text{ (L-ASSN)}$$

$$\frac{(\blacktriangleright tp) * Q \Longrightarrow (E_1 \mapsto \_) * P \qquad (\blacktriangleright tp) \ltimes ((E_1 \mapsto E_2) * P) \Longrightarrow G}{[\mathrm{I}], G, \mathrm{I} \vdash \{tp * Q\}[E_1] := E_2\{(E_1 \rightharpoonup E_2) \triangleright (tp * Q)\}} \text{ (S-ASSN)}$$

$$\frac{(\blacktriangleright tp) * Q \Longrightarrow (E_1 \mapsto \_) * (E_1' \mapsto \_) * P \qquad (\blacktriangleright tp) \ltimes ((E_1 \mapsto E_2) * (E_1' \mapsto E_2') * P) \Longrightarrow G}{[\mathrm{I}], G, \mathrm{I} \vdash \{tp * Q\}\langle [E_1] := E_2, [E_1'] := E_2' \rangle \{((E_1, E_1') \rightharpoonup (E_2, E_2')) \triangleright (tp * Q)\}} \text{ (S-ASSN-D)}$$

$$\frac{[\mathrm{I}], G, \mathrm{I} \vdash \{tp\} \iota \{tq\} \qquad \mathsf{sta}(\{tp, tq\}, R * \mathsf{Id}) \qquad \mathrm{I} \diamond \{R, G\}}{R, G, \mathrm{I} \vdash \{tp\} \iota \{tq\}} \text{ (ATOM-R)}$$

$$\frac{tp \Longrightarrow E \hookrightarrow \_}{[\mathrm{I}], G, \mathrm{I} \vdash \{tp\} r := [E]\{\exists \mathrm{X}. tp[\mathrm{X}/r] \wedge E[\mathrm{X}/r] \hookrightarrow r\}} \text{ (LOAD)}$$

$$\frac{}{R, G, \mathrm{I} \vdash \{tp\}\mathbf{skip}\{tp\}} \text{ (Skip)} \qquad \frac{}{R, G, \mathrm{I} \vdash \{tp\}\mathbf{mfence}\,\{\blacktriangleright tp\}} \text{ (MFNCE)}$$

$$\frac{\begin{array}{c}(\blacktriangleright tp) \Longrightarrow (E_1 \mapsto E) * P \qquad ((E_1 \mapsto E) * P) \ltimes P_{\mathrm{s}} \Longrightarrow G * \mathbf{true} \\ \exists \mathrm{X}. P_{\mathrm{s}}[\mathrm{X}/r] \wedge r = 1 \vee P_{\mathrm{f}}[\mathrm{X}/r] \wedge r = 0 \Longrightarrow Q \\ \text{where} \qquad P_{\mathrm{s}} = (E_1 \mapsto E_3) * P \wedge E_2 = E \qquad P_{\mathrm{f}} = (\blacktriangleright tp) \wedge E_2 \neq E\end{array}}{[\mathrm{I}], G, \mathrm{I} \vdash \{tp\} r := \mathsf{CAS}(E_1, E_2, E_3)\{Q\}} \text{ (CAS)}$$

$$\frac{R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\} \qquad \mathsf{sta}(tr, R' * \mathsf{Id}) \qquad \mathrm{I}' \diamond \{R', G'\} \qquad tr \Longrightarrow \mathrm{I}' * \mathsf{true} \qquad tr \Rrightarrow G' * \mathbf{true}}{R * R', G * G', \mathrm{I} * \mathrm{I}' \vdash \{tp * tr\}\mathrm{C}_1\{tq * tr\}} \text{ (FRAME)}$$

$$\frac{R, G, \mathrm{I} \vdash \{tp \wedge B\}\mathrm{C}_1\{tq\} \qquad R, G, \mathrm{I} \vdash \{tp \wedge \neg B\}\mathrm{C}_2\{tq\}}{R, G, \mathrm{I} \vdash \{tp\}\mathbf{if}\ B\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2\{tq\}} \text{ (IF)}$$

$$\frac{R, G, \mathrm{I} \vdash \{tp \wedge B\}\mathrm{C}\{tp\}}{R, G, \mathrm{I} \vdash \{tp\}\mathbf{while}\ B\ \mathbf{do}\ \mathrm{C}\{tp \wedge \neg B\}} \text{ (WHILE)} \qquad \frac{R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}_1\{tr\} \qquad R, G, \mathrm{I} \vdash \{tr\}\mathrm{C}_2\{tq\}}{R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}_1; \mathrm{C}_2\{tq\}} \text{ (SEQ)}$$

$$\frac{\begin{array}{c}tp \Longrightarrow tp' \qquad tq \Longrightarrow tq' \qquad R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\} \qquad \mathrm{I}' \diamond \{R', G'\} \\ R' \Longrightarrow R \qquad G \Longrightarrow G' \qquad (\mathsf{tp}' \vee tq') \Longrightarrow \lfloor \mathrm{I} \rfloor * \mathsf{true} \\ \mathsf{sta}(\{tp', tq'\}, R')\end{array}}{R', G', \mathrm{I}' \vdash \{tp'\}\mathrm{C}\{tq'\}} \text{ (CSQ)}$$

$$\frac{\begin{array}{c}\forall \mathrm{i}, \mathrm{j} \in [1, \mathrm{n}], (\mathrm{i} \neq \mathrm{j}).\ R_{\mathrm{i}}, G_{\mathrm{i}}, \mathrm{I} \vdash \{P_{\mathrm{i}} * Q_{\mathrm{i}}\}\mathrm{C}_{\mathrm{i}}; \mathbf{mfence}\,\{(P_{\mathrm{i}}' * Q_{\mathrm{i}}')\} \qquad G_{\mathrm{i}} \Longrightarrow R_{\mathrm{j}} \\ Q_{\mathrm{i}} \vee Q_{\mathrm{i}}' \Longrightarrow \mathrm{I} \qquad \mathsf{sta}(\{Q_{\mathrm{i}}, Q_{\mathrm{i}}'\}, R_{\mathrm{i}}) \qquad \mathrm{I} \diamond \{R_{\mathrm{i}}, G_{\mathrm{i}}\} \\ p \Longrightarrow \{\!|P_1|\!\}^{tid_1} * \cdots * \{\!|P_{\mathrm{n}}|\!\}^{tid_{\mathrm{n}}} * \left(\{\!|Q_1|\!\}^{tid_1} \wedge \cdots \wedge \{\!|Q_{\mathrm{n}}|\!\}^{tid_{\mathrm{n}}}\right) \\ \{\!|P_1'|\!\}^{tid_1} * \cdots * \{\!|P_{\mathrm{n}}'|\!\}^{tid_{\mathrm{n}}} * \left(\{\!|Q_1'|\!\}^{tid_1} \wedge \cdots \wedge \{\!|Q_{\mathrm{n}}'|\!\}^{tid_{\mathrm{n}}}\right) \Longrightarrow q\end{array}}{\mathrm{I} \vdash \{p\} tid_1.\mathrm{C}_1 \parallel \cdots \parallel tid_{\mathrm{n}}.\mathrm{C}_{\mathrm{n}}\{q\}} \text{ (Parallel)}$$

图 5.11: 逻辑推导规则

LOAD 是用于内存的读操作。我们知道，读指令 $\mathtt{r} := [E]$ 的操作语义要求 r 读到 E_1 的最新值。回顾我们在图 5.9中对 $\rightsquigarrow$ 的定义，$E \rightsquigarrow \mathfrak{n}$ 表示内存地址的 E 的最新值为 $\mathfrak{n}$, 即要么地址 E 所对应的内存的值为 $\mathfrak{n}$ 并且缓存中没有对它的写操作, 要么缓存中最新的对 E 的写操作为 $\mathfrak{n}$。所以 LOAD 规则的后断言表达了这一点，即 r 是读到了最新值。我们同时还要注意到，LOAD 规则的前提要求，E_1 是满足 tp 的内存中的有效地址，如果没有这条前提，我们没法保证 r 读到的值是有效地址。

Mfence 指令要求在执行前将清空缓存，因此根据我们的断言语义，我们可以在 MFENCE 规则中的后断言使用 $\blacktriangleright tp$ 来表达将满足前断言 tp 的机器状态的缓存清空以后所到达的机器状态。

接下来是 CAS 规则。初看起来它的前提断言很复杂，回忆我们在图 5.7中对 $\mathsf{CAS}(E_1, E_2, E_3)$ 指令的操作语义的定义，它要求几点：1. 执行这条语句前缓存为空。2. 执行完这条语句后缓存仍然为空。3. 如果地址 E_1 的值等于 E_2，则 CAS 执行成功，把 E_3 的值赋予地址 E_1 并返回 1 同时还会根据需要进行内存所有权的转移，否则 CAS 执行失败，直接返回 0。因此，我们的 CAS 的推倒规则即体现上面三点。首先，第一条前提要求线程首先将缓存清空才能执行 CAS 语句，并且地址 E_1 位于内存当中。第二条前提表示 CAS 指令如果执行成功，则共享内存前后状态需要满足保证条件 G。而 $\mathrm{P_s}$ 代表的是 CAS 指令执行成功以后的状态，$\mathrm{P_r}$ 代表的是 CAS 指令执行失败以后的状态。因此第三条前提说的就是，后条件满足这两种状态之一。注意到在这边我们后条件使用的 Q 是内存断言，没有缓存描述的部分，因为无论 CAS 执行成功与否，都会清空缓存。因此机器状态在执行完 CAS 指令之后，缓存应当为空。

下一条需要读者理解的规则是 FRAME 规则，这条规则来自于 [7], 可以允许我们的逻辑支持局部推理。即允许我们在某个上下文环境中下验证程序 C 以后，我们可以把这段证明重用到一个比之前验证时更大的上下文环境中 (tr 即额外加入的资源)。tr 包括了共享内存和局部内存两部分。因为 C 没有访问这块资源 (否则我们就推导不出 $R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\}$)，因此只要 tr 在新加入的依赖条件 R' 下稳定，那么 tr 在 C 执行前后都能保证合法。当然，我们同时要求新加入的不变量 I' 和 R', G' 有 fence 关系。和 [7] 中不同的是，由于我们的断言带有缓存描述，因此我们除了上述这些前提，我们还需要 $tr \Rrightarrow G' * \mathbf{true}$。这条前提的非形式化含义即，任何满足 tr 的机器状态，由于缓存中的写操作写入共享内存的而引起的状态变化，都需要满足 G'。形式化定义如下:

$$tp \Rrightarrow A \stackrel{\text{def}}{=} \forall \Delta, \Delta', \mathrm{I}.\ (\Delta \models_{\mathrm{I}} tp \wedge \Delta \rhd_{\mathrm{I}} \Delta') \Longrightarrow (\Delta, \Delta') \models A$$

CSQ 规则则可以允许我们弱化后条件和保证条件，并且强化前条件和依赖条件。同时，它要求新的前后条件需要在 R' 下是稳定的。并且新的依赖和保证条件可以被新的不变量 I 所 fence。

Parallel 是我们整个逻辑系统的顶级规则。它描述的是，在程序执行前，共享内存状态满足 $\left(\{\!\{Q_1\}\!\}^{tid_1} \wedge \cdots \wedge \{\!\{Q_n\}\!\}^{tid_n}\right)$；而每个线程 i 的局部状态满足 $\{\!\{P_1\}\!\}^{tid_1}$, 注意到，我们这边使用的是内存断言，即隐含了此时线程缓存为空的语义。那么，当整个程序执行完毕以后，并且，所有的线程都已经把缓存清空之后 (我们通过在第一条前提中，在语句 C_i 后面加入 **mfence** 来表示这一点)，则共享内存状态应当满足 $\left(\{\!\{Q_1'\}\!\}^{tid_1} \wedge \cdots \wedge \{\!\{Q_n'\}\!\}^{tid_n}\right)$, 并且每个线程 i 的局部状态满足 $\{\!\{P_1'\}\!\}^{tid_1}$。除此之外，我们通过依赖/保证条件，来保证，每个线程 i 的依赖条件 R_i 都能被其他线程 j 的保证条件 G_j 所满足。同时，每个线程关于共享内存状态的前后条件 $\{\!\{Q_1\}\!\}^{tid_1}$ 和 $\{\!\{Q_1'\}\!\}^{tid_1}$ 都应当在依赖条件下稳定，并且同时满足不变量 I 所描述的状态。

5.4　一些例子

在这一节中，我们将运用我们的逻辑系统给出对 Optimzed Implementation of Locks, Peterson's Lock, Simpson's Four Slot 算法以及 Concurrent GCD 算法在 TSO 内存模型下的正确性证明。在我们语言中，我们并没有变量的概念，所以我们只能用 [100], [200] 这样的表达方式来描述内存地址 100 和 200。这会导致我们的程序看起来丑陋并且违背直观。所以我们将在接下来的例子中使用 x,l,s,... 来代替这些代表地址的整数值，我们在这边强调的是，它们只是整数的符号代表，而不是变量。另外，下面的例子中，如果没有特殊说明，程序均以黑色显示，而断言以蓝色显示。

5.4.1　Optimzed Implementation of Locks 算法

在这个例子中，我们将展示我们的逻辑能够支持经过优化的 Spin Lock 算法的验证推导。图 5.12中是 SpinLock 的实现算法（用我们的语言）并且用 Spin Lock 来构造 push 和 pop 操作。图 5.12中，unlock 函数是经过优化的, 即它使用了非原子的写操作代替了 CAS 指令来进行锁资源的释放。我们假设在环境中有许多线程通过申请锁资源来对同一个共享栈 s 进行互斥的 pop 和 push 操作。当一个线程 ID 为 m 的线程想要对栈进行 push 或者 pop 操作时，它首先需要通过 spin lock 获得锁资源 l。当它获得锁资源以后 (意味着 l 的值此时是 m)，因为其他线程已经不能再对这个栈进行操作了 (因为无法获得锁资源)，所以此时我们可以把这个栈视为位于线程 m 的局部内存中，即线程将位于共享内存的栈通过内存所有权转移，转移到了自己的局部内存，并且在释放锁资源前都可以将这个栈视为私有资源来使用。当这个线程结束对栈的操作以后 (push 或者 pop) 将会释放锁资源，并且同时把栈放回共享内存中。我们要去证明如果 list(s) 在 push(或者 pop) 之前在共享内存上成立，那么在 push（或者 pop）操作后仍然成立。我们在图 5.12列出了线程 m 的 rely/guarantee 条件和不变量 I, 以及一些辅助定义。Guarantee 条件 G_m 说的是，当锁资源是 free 的时候，线程 m 将会通过对

l 赋值 m, 来获得锁资源 l，并且同时把栈放入它自己的局部内存当中; 并且当线程释放锁资源的时候，它会把栈再放回共享内存中。

证明大纲我们列在图 5.12中。容易知道，线程 m 的 Lock 函数的前条件 *tp* 首先要说明的是 l 要么是 free 的没有被任何线程获得 (即 l 的值为 0) 并且栈 s 位于共享内存中，要么 l 已经被其他线程获得，并且 s 不在共享内存中。当然，我们要注意到，由于经过优化的 Spin Lock 的 Unlock 函数最后并没有加入任何能够清空写缓存的 fence 语句，因此，除了前面两种情况，还有一种可能即，该锁资源之前已经被线程 m 获得，并且释放。但是释放锁资源的写操作仍然还处在线程 m 的缓存当中，因此，此时的状态应当是线程 m 的写缓存中有对锁资源 l 的写操作 (释放锁资源)，并且此时 l 在共享内存中的值是 m.

根据图 5.12, 我们知道函数 Push 的后条件是:

$$\mathsf{Push}_{post} = \left\{ \begin{array}{l} (x+1 \rightharpoonup r_1) \triangleright (s \rightharpoonup x) \triangleright (l \rightharpoonup 0) \triangleright l \mapsto m \\ \quad *(\exists y.(s \mapsto y * \mathsf{lseg}(y,0) * \exists v.x \mapsto v,\_) \wedge r_1 = y) \\ \vee (\exists n.\, n \neq m \wedge l \mapsto n) \end{array} \right\}$$

根据 ▶ 的定义, 我们有:

$$\blacktriangleright \mathsf{Push}_{post} \Rightarrow (l \mapsto 0 * \mathsf{list}(s)) \vee (\exists n.\, n \neq m \wedge l \mapsto n)$$

这意味着所有由 push 操作产生的写操作都被从写缓存中取出并按序写入内存当中，共享内存满足 $(l \mapsto 0 * \mathsf{list}(s)) \vee (\exists n.\, n \neq m \wedge l \mapsto n)$。类似的，我们可以通过 pop 的后条件得到相同的结论。

5.4.2　Peterson's Lock 算法

我们在图 5.13中展示了 Peterson's Lock 的算法。其中，我们把原始的 Peterson's Lock 语句用黑色标记，t 代表线程 ID, 它的值要么是 0 要么是 1(因为 Peterson's Lock 只使用双线程的竞争)。这个算法可以说是最有效率同时也是最为优雅的实现双线程互斥访问临界区的算法 [40]。然而，它在 TSO 内存模型下并不正确。原因在于在 TSO 模型中，每个线程都可以将它的前两条指令放入写缓存中 (即对 $f[tid]$ 和对 v 的写操作)。因此当它们进行读操作访问 $f[1-tid]$ 时，两个线程都读到 0，从而它们将会同时进入临界区进而违反了互斥访问。所以在 TSO 模型中，为了保证这个程序的正确，我们需要在对 v 写操作指令的后面添加 **mfence** 指令以用来让线程执行读操作之前确保缓存已经为空。此时这个添加了 **mfence** 指令的程序 (我们在图 5.13中用红色表示所添加的指令) 就能保证正确性。为了使用我们的逻辑系统去证明这个修改后的 Peterson's Lock 的正确性，我们还需要添加辅助变量 $a[0,1]$ 和 $b[0,1]$，同样，添加的辅助变量我们也用红色表示。根据我们修改后的程序，我们知道若 $b[tid] = 1$, 那么线程 t 处于临界区中。因此，我们需要证明 $b[0], b[1]$ 在程序执行的任何时刻都不能同时等于

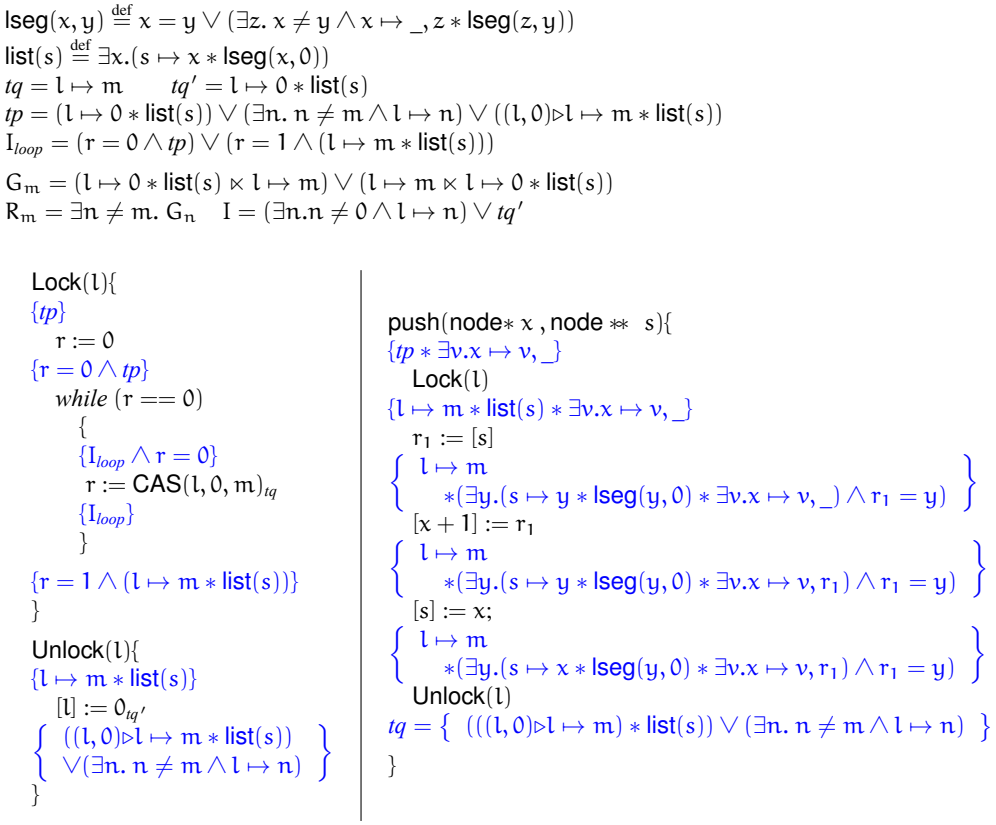


图 5.12: Spin Locks 算法证明证明

1，这就意味着在 TSO 模型下程序执行的任意时刻，Peterson's Lock 算法都能满足互斥访问。

每个线程的 Rely 和 Guarantee 条件，不变量 I 以及一些辅助断言定义如图 5.13中所示。我们使用一组不变量来构造 Rely 和 Guarantee 条件。这个证明也是相当直白，通过每条语句之后的后断言，我们可以很容易证明出 $\neg(b[0] = 1 \wedge b[1] = 1)$ 在程序执行的每一步中都成立。

5.4.3　Simpson's Four Slot 算法

在这一小节中，我们将展示如何使用我们的逻辑去验证 Simpson's Four Slot 算法，这将和 Ridge's 的工作 [15] 产生比较，因为它的工作中也验证了同一个程序在 TSO 模型上的正确性。我们在这边要强调的是，这个程序的验证结果并不能和 Ridge's 的工作有本质的区别，因为我们和他们工作的主要区别在于我们支持局部推理而他们不支持，然而这个程序的验证过程并不需要局部推理。我们在这边列出这个例子的原因只是因为我们想说明 Ridge 工作中能验证的例子我们也能验证，并不是说在这个例子中我们的逻辑能比他们展现出更大的优势。但是，在前面几个验证过程中带有内存所有权转移的例子中，我们的验证过程能做到比用他们逻辑更加的简单。Simpson's 算法如图 5.14中所示，同样，我们

$I_0 \stackrel{\text{def}}{=} a \mapsto \_, \_ * b \mapsto \_, \_ * f \mapsto \_, \_ * v \mapsto \_ \qquad I_1 \stackrel{\text{def}}{=} a[0,1], b[0,1], f[0,1], v \in \{0,1\}$
$I_2 \stackrel{\text{def}}{=} b[1-tid] = 1 \Rightarrow a[1-tid] = 1 \wedge f[1-tid] = 1 \wedge (a[tid] = 0 \vee [v] = tid)$
$I_4 \stackrel{\text{def}}{=} a[1-tid] = 1 \Rightarrow f[1-tid] = 1$
$I_3 \stackrel{\text{def}}{=} [v] = 1-tid \wedge a[tid] = 1 \Rightarrow f[1-tid] = 1 \wedge a[1-tid] = 1 \wedge b[1-tid] = 0$
$I_5 \stackrel{\text{def}}{=} a[tid] = X \wedge b[tid] = Y \wedge f[tid] = Z$
$I_G = I_0 \wedge I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5 \qquad G_{tid}, R_{1-tid} \stackrel{\text{def}}{=} \exists X, Y, Z.\ I_G \ltimes I_G \qquad I \stackrel{\text{def}}{=} I_0$
$tp_1 \stackrel{\text{def}}{=} I_0 \wedge I_1 \wedge a[tid] = 0 \wedge b[tid] = 0 \wedge f[tid] = 0 \wedge ((tp_7 \wedge [v] = 1-tid) \vee tp_5)$
$tp_2 \stackrel{\text{def}}{=} I_0 \wedge I_1 \wedge b[1-tid] = 0 \quad tp_3 \stackrel{\text{def}}{=} I_0 \wedge I_1 \wedge a[tid] = 1 \wedge b[tid] = 0 \wedge f[tid] = 1$
$tp_4 \stackrel{\text{def}}{=} ([v] = 1-tid \wedge b[1-tid] = 0)$
$tp_5 \stackrel{\text{def}}{=} a[1-tid] = 0 \wedge b[1-tid] = 0 \quad tp_6 \stackrel{\text{def}}{=} tp_5 \wedge [v] = tid \qquad tp_7 \stackrel{\text{def}}{=} a[1-tid] = 1 \wedge f[1-tid] = 1$
$I_{loop} = tp_3 \wedge ((tp_7 \wedge (([v] = tid \wedge r_1 = 1 \wedge r_2 = tid) \vee tp_4)) \vee (tp_5 \wedge (r_2 = tid \vee [v] = tid)))$

$\{tp_1\}$
$f[tid] := 1;$
$\{(f + tid \rightharpoonup 1) \triangleright tp_1\}$
$\langle [v] := tid, a[tid] := 1 \rangle;$
$\{ \begin{array}{l} ((f + tid \rightharpoonup 1) \triangleright ((v, a + tid) \rightharpoonup (tid, 1)) \triangleright tp_1) \\ \vee tp_3 \wedge ((tp_7 \wedge ([v] = tid \vee tp_4)) \vee tp_5) \end{array} \}$
mfence ;
$\{tp_3 \wedge ((tp_7 \wedge ([v] = tid \vee tp_4)) \vee tp_5)\}$
$r_1 := f[1-tid];$
$\left\{ \begin{array}{l} tp_3 \wedge ((tp_7 \wedge (([v] = tid \wedge r_1 = 1) \vee tp_4)) \\ \qquad \vee (tp_5 \wedge [v] = tid)) \end{array} \right\}$
$r_2 := v;$
$\{I_{loop}\}$
$while(r_1 == 1 \&\& r_2 == tid)$
$\{$
$\{r_1 = 1 \wedge r_2 = tid \wedge I_{loop}\}$
$r_1 := f[1-tid];$
$\{r_2 = tid \wedge I_{loop}\}$
$r_2 := v;$
$\{I_{loop}\}$
$\}$ //*be continued in the right side* ...

$\{tp_2\}$
$b[tid] := 1;$
$\{(b + tid \rightharpoonup 1) \triangleright tp_2\}$
...
critical region
...
$b[tid] := 0;$
$\{(b + tid \rightharpoonup 0) \triangleright (b + tid \rightharpoonup 1) \triangleright tp_2\}$
$a[tid] := 0;$
$\{(a + tid \rightharpoonup 0) \triangleright (b + tid \rightharpoonup 0) \triangleright (b + tid \rightharpoonup 1) \triangleright tp_2\}$
$f[tid] := 0;$
$\left\{ \begin{array}{l} (f + tid \rightharpoonup 0) \triangleright (a + tid \rightharpoonup 0) \triangleright (b + tid \rightharpoonup 0) \\ \qquad \triangleright (b + tid \rightharpoonup 1) \triangleright tp_2 \end{array} \right\}$

图 5.13: Peterson's Lock 证明

用黑色标示原始的代码颜色。这个算法也是同样设计用来保证单一读者和单一写者互斥访问资源。这个算法的优点是它不会阻塞，因为它维护了一个四元二维数组 $d[0..1, 0..1]$ 用来保存共享数据，即数组的每一维的长度都是 2，我们把每一元都称之为槽。因此写着和读者可以并发的访问不同的槽，而不会同时对同一个槽进行读写。在顺序一致性模型下，Simpson's 算法能够保证在临界区中读者和写者总是访问到不同的槽。

然而，这个程序在 TSO 模型下不再能保证这一点，因为当写者发出写操作指令以后，可能对 $sl[pair_w]$ 的写操作位于缓存里而不是已经写入内存中，因而不能保证正确性。因此，我们需要在写者代码之后加上 **mfence** 指令来保证在写者代码执行完毕优化缓存中没有其他写操作。同时我们还需要添加 **mfence** 指令到读者代码对 r 语句的写操作之后，来保证 $pair_r$ 的值等于 r。修改后的 Simpon 算法如图 5.14所示 (添加的代码用红色表示)。值得注意的是 $pair_w, pair_r, index_w, index_r$ 是线程局部地址，在我们的模型中，这些局部写操作

Writer:(w)
$\mathsf{pair}_w = !\mathsf{r}$
$\mathsf{index}_w = !\mathsf{sl}[\mathsf{pair}_w]$
$\mathsf{d}[\mathsf{pair}_w][\mathsf{index}_w] = w$
$\mathsf{sl}[\mathsf{pair}_w] = \mathsf{index}_w$
$\mathsf{la} = \mathsf{pair}_w$
mfence

Reader:
$\mathsf{pair}_r = \mathsf{la}$
$\mathsf{r} = \mathsf{pair}_r$
mfence
$\mathsf{index}_r = \mathsf{sl}[\mathsf{pair}_r]$
return $\mathsf{d}[\mathsf{pair}_r][\mathsf{index}_r]$

图 5.14: Simpson's Four Slot 算法

$$G_r = \exists A, B.(\neg(\mathsf{r} = \mathsf{la}) * \mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = A * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = B \ltimes \mathsf{r} = \mathsf{la} * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = A * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = B) \vee (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = A \ltimes \mathbf{emp}) \vee (\mathbf{emp} \ltimes \mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = B) \vee (\mathbf{emp} \ltimes \mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = A)$$

$$tq = \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true}$$
$$I = \mathsf{d}[0..1, 0..1] * \mathsf{sl}[0..1]$$

Writer:(w)
$\{tq\}$
$\mathsf{pair}_w = !\mathsf{r}$
$\{\exists A.\, \mathsf{pair}_w = !A * (\mathsf{r} = A * tq \vee \mathsf{la} = \mathsf{r} = !A * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_ * \mathsf{true})\}$
$\mathsf{index}_w = !\mathsf{sl}[\mathsf{pair}_w]$
$\{\exists A, B.\, \mathsf{index}_w = !B * \mathsf{pair}_w = !A * \mathsf{d}[!A][!B] = \_ * (\mathsf{r} = A * \mathsf{sl}[!A] = B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true} \vee \mathsf{la} = \mathsf{r} = !A * \mathsf{sl}[!A] = B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{true})\}$
$\mathsf{d}[\mathsf{pair}_w][\mathsf{index}_w] = w$
$\{\exists A, B.\, \mathsf{index}_w = !B * \mathsf{pair}_w = !A * \mathsf{d}[!A][!B] = w * (\mathsf{r} = A * \mathsf{sl}[!A] = B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true} \vee \mathsf{la} = \mathsf{r} = !A * \mathsf{sl}[!A] = B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{true})\}$
$\mathsf{sl}[\mathsf{pair}_w] = \mathsf{index}_w$
$\{\exists A, B.\, \mathsf{index}_w = !B * \mathsf{pair}_w = !A * (\mathsf{r} = A * \mathsf{sl}[!A] = !B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true} \vee \mathsf{la} = \mathsf{r} = !A * \mathsf{sl}[!A] = !B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true})\}$
$\mathsf{la} = \mathsf{pair}_w$
$\{(\mathsf{la} \rightharpoonup \mathsf{pair}_w) \triangleright (\exists A, B.\, \mathsf{index}_w = !B * \mathsf{pair}_w = !A * (\mathsf{r} = A * \mathsf{sl}[!A] = !B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee * \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true} \vee \mathsf{la} = \mathsf{r} = !A * \mathsf{sl}[!A] = !B * \mathsf{d}[!\mathsf{r}][\mathsf{sl}[!\mathsf{r}]] = \_ * \mathsf{d}[!\mathsf{r}][!\mathsf{sl}[!\mathsf{r}]] = \_ * (\mathsf{d}[\mathsf{r}][\mathsf{sl}[\mathsf{r}]] = \_ \vee \mathsf{d}[\mathsf{r}][!\mathsf{sl}[\mathsf{r}]] = \_) * \mathsf{true}))\}$
mfence
$\{tq\}$

图 5.15: Simpson's Four Slot 算法证明

都可以直接被写入内存而不用通过写缓存。由于我们处理局部内存的写操作和普通的分离逻辑一样，并且，在这个例子中，我们在所有的对共享单元的写操作后面都加上了 **mfence** 来清空缓存，因此，我们可以简单的修改一下 Shuling Wang 和 Xu Wang 在 ([41]) 的使用 RGSep 对这个例子的验证过程。因为证明的思路来自于 [41], 所以我们在图 5.15中只列出写者的证明大纲。

5.4.4　Concurrent GCD 算法

在这一小节当中，我们将展示如何通过我们的逻辑系统验证并发最大公约数算法 (concurrent GCD) 在 TSO 模型下的正确性。Concurrent GCD 是一个并发计算最大公约数的算法，左右线程并发使用辗转相除法进行计算直到得到最后的结果。我们使用这个例子来解释如何在断言中使用 $\mathsf{E}_1 \rightsquigarrow \mathsf{E}_2$ 来表达 E_1 的最新值是 E_2(即要么缓存中存在最新的对 E_1 写入 E_2 的写操作，或是缓存中没有这样的写操作，而内存地址中 E_1 的值是 E_2)。Concurrent GCD 算法如图 5.16中所示。左边线程读取 $[\mathsf{l}]$ 和 $[\mathsf{l}+1]$ 的值，但是仅仅在 $[\mathsf{l}] > [\mathsf{l}+1]$ 的情况下更新 $[\mathsf{l}]$。右边线程则是做相反的事，仅在 $[\mathsf{l}+1] > [\mathsf{l}]$ 的情况下更新 $[\mathsf{l}+1]$ 的值。内存单元 $[\mathsf{l}]$ 和 $[\mathsf{l}+1]$ 是被左右两个线程共享的。我们假设，初始时共享内存的状态是 $\mathsf{l} \mapsto \mathsf{M}, \mathsf{N}$. 我们这里使用 $\mathsf{l} \mapsto \mathsf{M}, \mathsf{N}$ 来作为 $\mathsf{l} \mapsto \mathsf{M} * \mathsf{l}+1 \mapsto \mathsf{N}$ 的缩写。我们想要证明如下的性质：在这个程序执行结束以后，$[\mathsf{l}]$ 和 $[\mathsf{l}+1]$ 的值是初始值 M 和 N 的最大公约数 ($\mathsf{gcd}(\mathsf{M},\mathsf{N})$)。我们同样在图 5.17列出 rely 和 guarantee 条件以及一些辅助断言。R_1 是对左边线程环境行为的假设。它说的是环境 (即右边线程) 不会修改 l 的值并且如果 $\mathsf{l}+1$ 的值比 l 的值大的话，他会减少 $\mathsf{l}+1$ 的值，同时保持新值的 GCD 和修改前一样。右边线程 R_2 的环境假设和 R_1 类似。我们同时还列出了左边线程的循环不变式 I_1. 它说的是当左边线程在共享内存 $\mathsf{l}+1$ 上读到值 Z 的时候，如果 l 的 “最新” 的值是 Y(在我们的断言中，我们可以很方便的用 $\mathsf{l} \rightsquigarrow \mathsf{Y}$ 表达这一点)，并且如果 Y 大于 Z, 那么，右边线程就不会往 $\mathsf{l}+1$ 写入新值。这个例子由于没有内存所有权转移，所以对于整个共享内存的不变式描述很容易就写出, 我们还是用 I 来表达。就如我们在上面 Peterson Lock 例子中做的哪样，根据对称性，我们只需要去证明左边线程即可。证明大纲如图 5.17所示。同样，根据 ▶ 的定义 (图 5.9), 我们可以有

$$\blacktriangleright tp_9 \Longrightarrow \exists \mathsf{U}.\mathsf{l} \mapsto \mathsf{U}, \mathsf{U} \wedge \mathsf{U} = \mathsf{gcd}(\mathsf{M},\mathsf{N})$$

所以我们就可以得出如下结论：并发 GCD 算法在 TSO 模型下是能够保证最后的计算结果是正确的。

5.5　逻辑可靠性证明

在这一节中，我们将证明我们的逻辑在 TSO 上的可靠性。由于我们的逻辑的语义是定义在 ATSO 之上的，因此我们把证明分成三步骤，首先，我们先证 TSO 模型是 ATSO 的精化 (refinement)，精化的非形式化描述是说对于任意一个程序，它在 TSO 模型下的行为，总是在 ATSO 下行为的子集。然后，我们将证明，我们的逻辑在 ATSO 上的可靠性。可靠性的非形式化描述可以理解为，任何满足使用我们逻辑断言所描述前条件的机器状态，在执行完语句以后，新的机器状态一定满足通过逻辑在这些语句上的推导所得到的后条件。也就是说，如

$\boxed{l \mapsto M, N}$

1 : $r_1 := [l];$	1 : $r_3 := [l+1];$
2 : $r_2 := [l+1];$	2 : $r_4 := [l];$
3 : while$(r_1 \neq r_2)$ do{	3 : while$(r_3 \neq r_4)$ do{
4 : if$(r_1 > r_2)$ then{	4 : if$(r_3 > r_4)$ then{
5 : $r_1 := r_1 - r_2$	5 : $r_3 := r_3 - r_4$
6 : $[l] := r_1;$	6 : $[l+1] := r_3;$
7 : }	7 : }
8 : $r_2 := [l+1];$	8 : $r4 := [l];$
9 : }	9 : }

$\boxed{\exists U. l \mapsto U, U \wedge U = \mathsf{gcd}(M, N)}$

图 5.16: Concurrent GCD 算法

$p_1 \stackrel{\text{def}}{=} l \mapsto Y, Z \wedge Y < Z$
$p_1' \stackrel{\text{def}}{=} \exists Z'.\, l_1 \mapsto Y, Z' \wedge Z' < Z \wedge \mathsf{gcd}(Y, Z) = \mathsf{gcd}(Y, Z')$
$p_2 \stackrel{\text{def}}{=} l \mapsto Y, Z \wedge Y > Z$
$p_2' \stackrel{\text{def}}{=} \exists Y'.\, l \mapsto Y', Z \wedge Y' < Y \wedge \mathsf{gcd}(Y, Z) = \mathsf{gcd}(Y', Z)$
$R_1, G_2 \stackrel{\text{def}}{=} \exists Y, Z.\, p_1 \ltimes p_1' \qquad R_2, G_1 \stackrel{\text{def}}{=} \exists Y, Z.\, p_2 \ltimes p_2'$
$tp \stackrel{\text{def}}{=} \exists Z.\, l \mapsto M, Z \wedge \mathsf{gcd}(M, Z) = \mathsf{gcd}(M, N)$
$I_l \stackrel{\text{def}}{=} \exists wl, Z'.\, wl \triangleright l \mapsto \_, Z' \wedge Z \geqslant Z' \wedge l \leadsto Y \wedge (Y \geqslant Z \Rightarrow Z = Z') \wedge \mathsf{gcd}(Y, Z') = \mathsf{gcd}(M, N)$
$I \stackrel{\text{def}}{=} l \mapsto \_, \_$

$tp_0 : \{tp\}$
$r_1 := [l];$
$tp_1 : \{r_1 = M \wedge tp\}$
$r_2 := [l+1];$
$tp_2 : \{\exists Y, Z.\, r_1 = Y \wedge r_2 = Z \wedge I_l\}$#*Here* $Y = M$, *and* $wl =$ **nil** *in the invariant* I
while$(r_1 \neq r_2)$ do{
$tp_3 : \{r_1 \neq r_2 \wedge \exists Y, Z.\, r_1 = Y \wedge r_2 = Z \wedge I_l\}$
  if$(r_1 > r_2)$ then{
$tp_4 : \{r_1 > r_2 \wedge tp_3\}$
    $r_1 := r_1 - r_2$
$tp_5 : \{\exists Y, Z.\, r_1 = Y - Z \wedge r_2 = Z \wedge Y > Z * I_l\}$
    $[l] := r_1;$
$tp_6 : \{\exists Y, Z.\, r_1 = Y - Z \wedge r_2 = Z \wedge Y > Z * I_l\}$
  }
$tp_7 : \{\exists Y, Z.\, r_1 = Y \wedge r_2 = Z * I_l\}$
  $r_2 := [l+1];$
$tp_8 : \{\exists Y, Z.\, r_1 = Y \wedge r_2 = Z * I_l\}$
}
$tp_9 : \{\exists Y, Z.\, r_1 = Y \wedge r_2 = Z \wedge Y = Z * I_l\}$

图 5.17: Concurrent GCD 算法证明 (左边线程)

果能够用我们的逻辑推导描述出一段程序的行为，则当这段程序实际在 ATSO 的抽象机上运行不会偏离用我们的逻辑所描述的行为。综上两点，我们就能推出，我们逻辑在 TSO 上的可靠性。

5.5.1 TSO 是 ATSO 的精化

在这一子节中，我们将使用模拟关系来证明精化关系。我们首先定义 TSO 机器状态和 ATSO 机器状态的等价性:

定义 5.5.1 (Buffer Flushed)**.**

$$\blacktriangleright\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma' & \textit{if } (\mathbf{skip},\sigma) \rightsquigarrow^* (\mathbf{skip},\sigma') \textit{ and } \forall tid.\ \sigma'.thds(tid).buf = \mathbf{nil} \\ \textit{undef} & \textit{if there is no such } \sigma' \end{cases}$$

$$\blacktriangleright_{\mathrm{I}}\Sigma \stackrel{\text{def}}{=} \begin{cases} \Sigma' & \textit{if } \mathrm{I} \vdash (\mathbf{skip},\Sigma) \rightsquigarrow^* (\mathbf{skip},\Sigma') \textit{ and } \forall tid.\ \Sigma'.thds(tid).buf = \mathbf{nil} \\ \textit{undef} & \textit{if there is no such } \Sigma' \end{cases}$$

定义 5.5.2 (State Equivalence)**.**

$$\mathsf{local}(\Sigma) \stackrel{\text{def}}{=} \biguplus_{tid}(\Sigma.T(tid).m)$$

$$\begin{aligned} \sigma \approx_{\mathrm{I}} \Sigma \stackrel{\text{def}}{=}\ & (\blacktriangleright\sigma).m = ((\blacktriangleright_{\mathrm{I}}\Sigma).m \uplus \mathsf{local}(\blacktriangleright_{\mathrm{I}}\Sigma)) \\ & \textit{and } (\forall tid.\ \sigma.thds(tid).rf = \Sigma.T(tid).rf) \\ & \textit{and } (\forall tid.\ \Sigma.T(tid).buf \subseteq \sigma.thds(tid).buf) \end{aligned}$$

$$\begin{aligned} \sigma \equiv \Sigma \stackrel{\text{def}}{=}\ & \sigma.m = \Sigma.m \uplus \mathsf{local}(\Sigma) \\ & \textit{and } (\forall tid.\ \sigma.thds(tid).rf = \Sigma.T(tid).rf) \\ & \textit{and } (\forall tid.\ \Sigma.T(tid).buf = \sigma.thds(tid).buf = \mathbf{nil}) \end{aligned}$$

引理 5.5.1. *if* $\sigma \approx_{\mathrm{I}} \Sigma$*, and* $\forall tid.\ \Sigma.T(tid).buf = \sigma.thds(tid).buf = \mathbf{nil}$*, then* $\sigma \equiv \Sigma$。

证明. 我们可以直接通过展开 $\approx_{\mathrm{I}}$ 和 $\equiv$ 的定义来证明。

□

引理 5.5.2 (精化关系)**.** 对于任意的 σ, Σ, P, 若存在 I 使得 $\sigma \approx_{\mathrm{I}} \Sigma$ 并且 $\mathrm{I} \vdash (P,\Sigma) \not\rightsquigarrow^* \mathsf{abort}$, 那么对于任意的 σ', P', 若 $(P,\sigma) \rightsquigarrow (P',\sigma')$, 则存在 Σ' 使得 $\mathrm{I} \vdash (P,\Sigma) \rightsquigarrow^* (P',\Sigma')$ 并且 $\sigma' \approx_{\mathrm{I}} \Sigma'$。

证明. 证明过程我们放在附录 C。 □

推论 5.5.1. 对于任意的 σ, Σ, P, 如果 $\sigma \equiv \Sigma$ 并且存在 I 使得 $\mathrm{I} \vdash (P,\Sigma) \not\rightsquigarrow^* \mathsf{abort}$, 那么对于任意的 σ', 若 $(P,\sigma) \rightsquigarrow^* (\mathbf{skip},\sigma')$ 并且 $\forall tid.\ \sigma'.thds(tid).buf = \mathbf{nil}$, 则存在 Σ' 使得 $\mathrm{I} \vdash (prog,\Sigma) \rightsquigarrow^* (\mathbf{skip},\Sigma')$ 并且 $\sigma' \equiv \Sigma'$。

证明. 我们可以在转移状态 $(prog,\sigma) \rightsquigarrow^* (\mathbf{skip},\sigma')$ 的步数上进行递归并且运用引理5.5.2来证明。 □

5.5.2　逻辑系统在 ATSO 上的可靠性

定义 5.5.3. $\mathcal{R},\mathcal{G},\mathrm{I} \models \{tp\}\mathrm{C}\{tq\}$ 当且仅当，对于任意的 Δ 使得 $\Delta \models_{\mathrm{I}} tp$ 并且 $\mathcal{R} \models_{\mathrm{I}} \Delta{:}\mathcal{G}$，我们都有 $\mathcal{R},\mathrm{I} \models_{\mathfrak{n}} (\mathrm{C},\Delta){:}(\mathcal{G},tq)$ 对于任意 $\mathfrak{n}$ 成立。

定义 5.5.4. $\mathcal{R},\mathrm{I} \models_0 (\mathrm{C},\Delta){:}(\mathcal{G},tq)$ 总是成立的。$\mathcal{R},\mathrm{I} \models_{\mathfrak{n}+1} (\mathrm{C},\Delta){:}(\mathcal{G},tq)$ 当且仅当下面几点为真：

- $\mathcal{R} \models_{\mathrm{I}} \Delta{:}\mathcal{G}$；
- 若 $\mathrm{C} = \textbf{skip}$，则 $\Delta \models_{\mathrm{I}} tq$；
- 若 $\mathrm{C} \neq \textbf{skip}$，则存在 C' 和 Δ' 使得 $\mathrm{I} \vdash (\mathrm{C},\Delta) \rightsquigarrow (\mathrm{C}',\Delta')$，$(\Delta.m_s,\Delta'.m_s) \in \mathcal{G}$，并且 $\mathcal{R},\mathrm{I} \models_{\mathfrak{n}} (\mathrm{C}',\Delta'){:}(\mathcal{G},tq)$；
- 对于任意的 m' 使得 $(\Delta.m_s,m') \in \mathcal{R}$，我们都有 $\mathcal{R},\mathrm{I} \models_{\mathfrak{n}} (\mathrm{C},(m',\Delta.m_l,\Delta.rf,\Delta.buf)){:}(\mathcal{G},tq)$；

定义 5.5.5. $\mathcal{R} \models_{\mathrm{I}} (m_s,m_l,rf,buf) : \mathcal{G}$ 当且仅当要么 $buf = \textbf{nil}$，或者对于任意的 $m_s{}'$ 使得 $(m_s,m_s{}') \in \mathcal{R}^*$，则存在 $m_s{}'',m_l{}',rf'$ 和 buf' 使得 $(m_s{}',m_l,rf,buf) \rhd_{\mathrm{I}} (m_s{}'',m_l{}',rf',buf')$，$(m_s{}',m_s{}'') \in \mathcal{G}$，并且 $\mathcal{R} \models_{\mathrm{I}} (m_s{}'',m_l{}',rf',buf'){:}\mathcal{G}$。

推论 5.5.2. 若 $\mathcal{R} \models_{\mathrm{I}} (m_s,m_l,rf,buf){:}\mathcal{G}$，则 $[\![[\mathrm{I}]]\!] \models_{\mathrm{I}} (m_s,m_l,rf,buf){:}\mathcal{G}$

证明. 直接通过定义5.5.5即可证明。 □

推论 5.5.3. 若 $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$，并且 $\Delta.buf \neq \textbf{nil}$，则存在 Δ' 使得 $\Delta \rhd_{\mathrm{I}} \Delta'$，$(\Delta.m_s,\Delta'.m_s) \in \mathcal{G}$，并且 $\mathcal{R} \models_{\mathrm{I}} \Delta'{:}\mathcal{G}$。

证明. 我们知道对于任意的 $m_s{}'$，如果它使得 $(\Delta.m_s,m_s{}'') \in \mathcal{R}^*$ 成立的话，则存在 $m_s{}',m_l{}',rf'$ 和 buf' 使得 $(m_s{}'',m_l,rf,buf) \rhd_{\mathrm{I}} (m_s{}',m_l{}',rf',buf')$，$(m_s{}'',m_s{}') \in \mathcal{G}$，以及 $\mathcal{R} \models_{\mathrm{I}} (m_s{}',m_l{}',rf',buf') : \mathcal{G}$ 成立 (我们可以通过定义 5.5.5展开 $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$，以 $\Delta.buf \neq \textbf{nil}$ 来得到这一点)。然后我们令 $m_s{}'' = \Delta.m_s$，以及 $\Delta' = (m_s{}',m_l{}',rf',buf')$，于是我们就能得出结论。 □

推论 5.5.4. 若 $\mathcal{R} \models_{\mathrm{I}} (m_s,m_l,rf,buf){:}\mathcal{G}$ 并且对于任意的 $m_s{}'$ 使得 $(m_s,m_s{}') \in \mathcal{R}^*$，则 $\mathcal{R} \models_{\mathrm{I}} (m_s{}',m_l,rf,buf){:}\mathcal{G}$。

证明. 我们可以根据定义5.5.5来证明这一点。 □

推论 5.5.5. 若 $\Delta \rhd_{\mathrm{I}} \Delta'$ 并且 $\Delta \rhd_{\mathrm{I}} \Delta''$，则我们有 $\Delta' = \Delta''$。

证明. 根据 $\rhd_{\mathrm{I}}$ 的操作语义，我们可以直接得出结论。 □

推论 5.5.6. 如果 $\Delta \models_{\mathrm{I}} tp$ *and* $\Delta \rhd_{\mathrm{I}} \Delta'$，并且我们有 $\Delta' \models_{\mathrm{I}} tp$。

证明. 我们之前在引理5.2.1中证明我们逻辑的断言是基于缓存稳定的，因此我们可以很简单的得出结论。 □

推论 5.5.7. 对于任意的 Δ 使得 $\Delta \models_{\mathrm{I}} tp$，$[\![R]\!] \models_{\mathrm{I}} \Delta{:}\mathcal{G}$，$[\![[\mathrm{I}]]\!],\mathrm{I} \models_{\mathfrak{n}} (\iota,\Delta){:}(\mathcal{G},tq)$，$\mathsf{sta}(\{tp,tq\},R)$ 并且 $\mathrm{I} \diamond R$ 则 $[\![R]\!],\mathrm{I} \models_{\mathfrak{n}} (\iota,\Delta){:}(\mathcal{G},tq)$。

证明. 对 $\mathfrak{n}$ 进行归纳假设

- Base: $\mathfrak{n} = 0$. 这种情况是平凡的，直接根据定义可证。

- Inductive: $\mathtt{n} = \mathtt{j} + 1$. 根据定义5.5.4我们需要证明以下四个子目标：
  - $[\![R]\!] \models_{\mathrm{I}} \Delta : \mathcal{G}$;
    根据前提直接可得。
  - 若 $\iota = \textbf{skip}$, 则 $\Delta \models_{\mathrm{I}} tq$;
    展开 $[\![[\mathrm{I}]]\!], \mathrm{I} \models_{\mathtt{n}} (\iota, \Delta) : (\mathcal{G}, tq)$ 可得。
  - 若 $\iota \neq \textbf{skip}$, 则存在 ι' 和 Δ' 使得 $\mathrm{I} \vdash (\iota, \Delta) \rightsquigarrow (\iota', \Delta'), (\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $[\![R]\!], \mathrm{I} \models_{\mathtt{j}} (\iota', \Delta') : (\mathcal{G}, tq)$;
    同上，根据展开 $[\![[\mathrm{I}]]\!], \mathrm{I} \models_{\mathtt{n}} (\iota, \Delta) : (\mathcal{G}, tq)$，我们有存在 ι' 和 Δ' 使得 $\mathrm{I} \vdash (\iota, \Delta) \rightsquigarrow (\iota', \Delta'), (\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $[\![[\mathrm{I}]]\!], \mathrm{I} \models_{\mathtt{n}} (\iota', \Delta') : (\mathcal{G}, tq)$;
    根据归纳假设，我们有 $[\![R]\!], \mathrm{I} \models_{\mathtt{n}} (\iota', \Delta') : (\mathcal{G}, tq)$。得证。
  - 对于任意的 m' 使得 $(\Delta.m_s, m') \in [\![R]\!]$, 我们都有
    $[\![R]\!], \mathrm{I} \models_{\mathtt{n}} (\iota, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tq)$;
    因为 $\mathsf{sta}(\{\mathtt{tp}, \mathtt{tq}\}, \mathtt{R})$, 所以我们有 $(m', \Delta.m_l, \Delta.rf, \Delta.buf) \models \mathtt{tp}$
    根据推论 5.5.4, 由于有 $[\![[\mathrm{I}]]\!] \models_{\mathrm{I}} \Delta : \mathcal{G}$，则我们有
    $[\![[\mathrm{I}]]\!] \models_{\mathrm{I}} (m', \Delta.m_l, \Delta.rf, \Delta.buf) : \mathcal{G}$。根据归纳假设可得结论。

□

定义 5.5.6 (环境与写缓存对状态的耦合影响)**.**

$$
\begin{array}{rcl}
\rhd_{\mathrm{I}}^{\mathcal{R}} (m_s, m_l, rf, buf) & = & \{(m_s', m_l', rf', buf') \mid (m_s, m_l, rf, buf) \rhd_{\mathrm{I}} (m_s', m_l', rf', buf') \\
 & & \quad or\ \exists m.\ (m_s, m_l, rf, buf) \rhd_{\mathrm{I}} (m, m_l', rf', buf') \wedge (m', m_s') \in \mathcal{R}^*\} \\
\blacktriangleright_{\mathrm{I}}^{\mathcal{R}} (m_s, m_l, rf, buf) & = & \{(m_s', m_l', rf', \textbf{nil}) \mid (m_s', m_l', rf', \textbf{nil}) \in (\rhd_{\mathrm{I}}^{\mathcal{R}} (m_s, m_l, rf, buf)) \\
 & & \quad or\ \exists \Delta.\ \Delta \in (\rhd_{\mathrm{I}}^{\mathcal{R}} (m_s, m_l, rf, buf)) \wedge (m_s', m_l', rf', \textbf{nil}) \in (\blacktriangleright_{\mathrm{I}}^{\mathcal{R}} \Delta)\}
\end{array}
$$

推论 5.5.8. *若 $\Delta \models_{\mathrm{I}} tp$, $\mathsf{sta}(\mathtt{tp}, R * \mathrm{Id})$, $\mathcal{R} = [\![R * \mathsf{Id}]\!]$ 并且 $\Delta' \in (\blacktriangleright_{\mathrm{I}}^{\mathcal{R}} \Delta)$, 则我们有 $\Delta' \models tp$。*

证明. 我们可以通过推论5.5.6和定义5.2.3来得出这一点。 □

引理 5.5.3 (隔离性)**.** *如果 $\mathcal{R} \models_{\mathrm{I}} (m_s, m_l \uplus \{a \mapsto v\}, rf, buf) : \mathcal{G}$, $\blacktriangleright_{\mathrm{I}} (m_s, m_l \uplus \{a \mapsto v\}, rf, buf) = (m_{s1}, m_{l1}, rf_1, \textbf{nil})$ 并且 $a \in \mathsf{dom}(m_{l1})$ 对于任意的 v', 我们有 $\mathcal{R} \models_{\mathrm{I}} (m_s, m_l \uplus \{a \mapsto v'\}, rf, buf) : \mathcal{G}$ 恒成立。*

证明. 根据写缓存 *buf* 的长度进行归纳假设:

- $buf = \textbf{nil}$. 这种情况根据定义可以直接得出结论。
- 假设 $buf = buf'\texttt{++}u$。

  根据定义 5.5.5, 我们需要证明对于所有的 m_s' 使得 $(m_s, m_s') \in \mathcal{R}^*$, 存在 m_s'', m_l', rf' 和 buf' 使得 $(m_s', m_l[a \mapsto v]', rf, buf) \rhd_{\mathrm{I}} (m_s'', m_l', rf', buf')$, $(m_s', m_s'') \in \mathcal{G}$, 以及 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l', rf', buf') : \mathcal{G}$ 成立。

  由于我们有 $\blacktriangleright_{\mathrm{I}} (m_s, m_l[a \mapsto v], \mathtt{rf}, buf) = (m_{s1}, m_{l1}, rf_1, \textbf{nil})$ 以及 $a \in \mathsf{dom}(m_{l1})$, 所以我们知道 u 不会修改内存地址 a or 把它转移到共享内存中。因此我们有 $m_l' = m_l \uplus \{a = v'\}$。

  再次根据定义 5.5.5，我们知道对于任意的 m_s'' 使得 $(m_s, m_s'') \in \mathcal{R}^*$, 存在 m_s''', m_l'', rf'' 以及 buf'' 使得 $(m_s'', m_l \uplus \{a \mapsto v\}, rf, buf) \rhd_{\mathrm{I}}$ $(m_s''', m_l'', rf'', buf'')$, $(m_s'', m_s''') \in \mathcal{G}$, 以及 $\mathcal{R} \models_{\mathrm{I}} (m_s''', m_l'', rf'', buf'') : \mathcal{G}$ 成立。如我们上面分析的那样，我们知道 $m_l'' = m_l \uplus \{a = v\}$。我们可以简单

地得出 $\blacktriangleright_{\mathrm{I}}(m_s''', m_l'', rf, buf) = (m_{s1}', m_{l1}, rf_1, \mathbf{nil})$ 和 $a \in \mathsf{dom}(m_{l1})$. 根据归纳假设我们有 $\mathcal{R} \models_{\mathrm{I}} (m_s''', m_l \uplus \{a \mapsto v'\}, rf, buf) : \mathcal{G}$. 并且 $m_s''' = m_s''$ 显然成立，因此我们有 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l \uplus \{a \mapsto v'\}, rf, buf) : \mathcal{G}$。所以我们现在能得出结论。

□

引理 5.5.4. *若 $\mathcal{R} \models_{\mathrm{I}} (m_s, m_l, rf, buf) : \mathcal{G}$, $(m_{s1}, m_{l1}, rf_1, \mathbf{nil}) \in (\blacktriangleright_{\mathrm{I}}^{\mathcal{R}}(m_s, m_l, rf, buf))$, $(m_{s1}, m_{s1}[a \mapsto v]) \in \mathcal{G}$ 并且 $a \in \mathsf{dom}(m_{s1})$, 则 $\mathcal{R} \models_{\mathrm{I}} (m_s, m_l, rf, (a, v) :: buf) : \mathcal{G}$。*

证明. 同样根据 *buf* 的长度进行归纳假设：

- Base case: $buf = \mathbf{nil}$;

  我们需要证明对于任意的 m_s' 使得 $(m_s, m_s') \in \mathcal{R}^*$, 则存在 m_s'', m_l', rf' 以及 buf' 使得 $(m_s', m_l, rf, (a, v) :: \mathbf{nil}) \rhd_{\mathrm{I}} (m_s'', m_l', rf', \mathbf{nil})$, $(m_s', m_s'') \in \mathcal{G}$, 和 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l', rf', \mathbf{nil}) : \mathcal{G}$ 成立。因为 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l', rf', \mathbf{nil}) : \mathcal{G}$ 成立是平凡的，所以我们能立即得出结论。

- 假设 $buf = buf' \mathbin{+\!\!+} u$; 根据定义 5.5.5, 我们需要证明对于任意的 m_s' 使得 $(m_s, m_s') \in \mathcal{R}^*$, 则存在 m_s'', m_l', rf' 和 buf' 使得 $(m_s', m_l, rf, (a, v) :: buf) \rhd_{\mathrm{I}} (m_s'', m_l', rf', (a, v) :: buf')$, $(m_s', m_s'') \in \mathcal{G}$, 以及 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l', rf', (a, v) :: buf') : \mathcal{G}$。

  根据定义5.5.5展开 $\mathcal{R} \models_{\mathrm{I}} (m_s, m_l, rf, buf) : \mathcal{G}$，我们有 $\mathcal{R} \models_{\mathrm{I}} (m_s'', m_l', rf', buf') : \mathcal{G}$。然后我们就能使用归纳假设得出证明。

□

引理 5.5.5. *若 $\mathcal{R}, \mathrm{I} \models_{\mathtt{n}} (\mathrm{C}, \Delta) : (\mathcal{G}, tq)$ 对于任意的 $\mathtt{n}$ 均成立, 则对于任意的 C', Δ' 使得 $\mathrm{I} \vdash (\mathrm{C}, \Delta) \rightsquigarrow (\mathrm{C}', \Delta')$, 我们都有 $\mathcal{R}, \mathrm{I} \models_{\mathtt{n}} (\mathrm{C}', \Delta') : (\mathcal{G}, tq)$ 对于任意的 $\mathtt{n}$ 都成立。*

证明. 根据 $\mathcal{R}, \mathrm{I} \models_{\mathtt{n}} (\mathrm{C}, \Delta) : (\mathcal{G}, tq)$ 的定义展开，我们能够立即得出结论。 □

引理 5.5.6. *若 $R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\}$, $\mathcal{R} = [\![R]\!]$, $\mathcal{G} = [\![G]\!]$, 则 $\mathcal{R}, \mathcal{G}, \mathrm{I} \models \{tp\}\,\mathrm{C}\,\{tq\}$。*

证明. 我们把证明放在附录 D。 □

定义 5.5.7. *$\models_{\mathrm{I}} \{p\}prog\{q\}$ 对于任意的 Σ 使得 $\Sigma \models_{\mathrm{I}} p$ 并且 $\mathrm{I} \vdash (prog, \Sigma) \not\rightsquigarrow {}^*\mathsf{abort}$, 若 $(prog, \Sigma) \rightsquigarrow^* (\mathbf{skip}, \Sigma')$ 并且 $\forall \mathtt{i}.(\mathtt{i} \in \Sigma'.tq) \Longrightarrow (\Sigma'.tq(\mathtt{i}).buf = \mathbf{nil})$ 则 $\Sigma' \models_{\mathrm{I}} q$。*

引理 5.5.7. *若 $\mathrm{I} \vdash \{p\}prog\{q\}$, 则 $\models_{\mathrm{I}} \{p\}prog\{q\}$。*

证明. 根据引理5.5.6和 Parallel Rule, 得证。 □

定义 5.5.8 (断言在 TSO 上的语义)**.**

$$(m, thds) \models_{\mathsf{TSO}} p \stackrel{\text{def}}{=} \exists \Sigma, \mathrm{I}.\ \Sigma \models_{\mathrm{I}} p \wedge (m, thds) \equiv \Sigma$$

定义 5.5.9 (TSO Semantics)**.** *$\models_{\mathsf{TSO}} \{p\}prog\{q\}$ 当且仅当对于任意的 σ 使得 $\sigma \models_{\mathsf{TSO}} p$ 并且 $\forall tid.\ \sigma'.thds(tid).buf = \mathbf{nil}$, 则 $(prog, \sigma) \not\rightsquigarrow {}^*\mathsf{abort}$, 并且对于任意的 σ', 若 $(prog, \sigma) \rightsquigarrow^* (\mathbf{skip}, \sigma')$, 并且 $\forall tid.\ \sigma'.thds(tid).buf = \mathbf{nil}$, 则我们有 $\sigma' \models_{\mathsf{TSO}} q$。*

定理 5.5.1 (LRGTSO 在 TSO 上的可靠性)**.** *若 $\mathrm{I} \vdash \{p\}prog\{q\}$ 则 $\models_{\mathsf{TSO}} \{p\}prog\{q\}$。*

证明. 根据引理5.5.2, 我们知道 TSO 是 ATSO 的精化，根据引理5.5.2我们知道 LRGTSO 在 ATSO 上的可靠性，根据定义5.5.9展开 $\models_{\mathsf{TSO}} \{\mathrm{p}\}prog\{\mathrm{q}\}$, 我们可以证明此定理。 □

5.6　相关工作和总结

Ridge [15] 提出了一种用于 X86-TSO 的 R-G 逻辑，这项工作和我们的工作最为接近。他直接运用了 HOL(Hoare Logic) 断言来描述程序状态，并且也支持相当有表达性的缓存断言。他们的逻辑在 TSO 上证明了 Simpson's Four Slot 算法的正确性。他们的工作和我们工作的关键区别在于他们的逻辑缺乏局部推理。因此在他们的逻辑当中，并没有作用在缓存断言上的 Separating Conjunction，也没有内存所有权的转移。并且，他们的工作也没有清楚的说明，怎么支持在验证中添加的辅助变量来进行证明 (在有些证明当中，辅助变量的存在是不可或缺的)。

Wehrman 和 Berdine [42] 的工作中展示了如何拓展并发分离逻辑并使之运用于 TSO 弱内存模型之上。他们提出了一种叫做 Sequential Conjunction 的方法使得断言不仅能够记录缓存中的写操作之间的顺序，同业也能记录这些写操作和已经实际写入堆中的写操作之间的顺序。为了能够定义 Separating Conjunction 和内存所有权转移，他们不得不在断言语言中引入两种不同的权限，这导致了他们的逻辑比起我们的复杂很多。同时在他们文章发表的时候，他们逻辑的可靠性并没有被证明。根据其作者所说，由于使用了不同权限的断言，该逻辑系统可能会不具备可靠性。

Vafeiadis 和 Naraya 在 [16] 中发表了 relaxed separation lgic(RSL)，这是一个用于 C11 内存模型的程序逻辑，而 C11 内存模型和 TSO 模型差别很大。我们的工作和他们之间另一个关键区别是 RSL 是设计只用来验证无数据竞争 (DRF) 的程序，尽管 C11 内存模型中对 DRF 的定义和传统的有很大区别。而我们的工作则是用来同时应对无数据竞争的程序和有数据竞争的程序。当然，我们的工作和他们的本质区别是来自于 TSO 内存模型和 C11 内存模型设计理念的不同，在 C11 模型中并不关心那些产生数据竞争的程序, 因为这样的程序的语义在 C11 模型中属于未定义的，它们可能由于不同的编译器不同的硬件产生不同的行为。Aaron Turon 等人在 [43] 发表了 GPS(Ghost state, Protocols, Separation Logic) 程序逻辑, 如其名字所示，它是第一个在弱内存模型上支持这三种现代验证技术的程序逻辑。和我们的逻辑一样，它同样支持 rely-guarantee 推理以及 ownership transfer，但是它仍然是设计用于 C11 内存模型的程序逻辑，因此仍然会和我们的工作有着本质的区别。

当然，还有很多工作是通过自动化的 fence 插入或者删除来在弱内存模型中产生正确同步的程序，比如 [44–47]. 他们的工作目标事实上和我们目前的工作是正交的，即我们的目标是设计一个富有表达力的能够验证任何给定的程序（这些程序可能是手工优化的，并且是有数据竞争的）在 TSO 上是否正确的逻辑系统。而他们的工作则是自动化的验证经过插入同步语句而正确同步的程序（这些没有数据竞争的程序在弱内存模型和 SC 模型上行为一致，因此可以通过

已有的在 SC 模型上自动化验证的方法来验证)。我们在本章中主要集中在设计逻辑，自动化验证的部分我们将其留给之后的工作。

第六章　结论

为了确保能够写出正确的并发程序，程序要必须准确理解并发程序的运行模式，或者说程序语义。而内存一致性模型描述了并发程序访问内存的方式。在现实世界的实现中，为了允许编译器和底层硬件的各种优化，这些软硬件都是基于弱内存模型的假设。本文主要在弱内存模型上进行研究，大致分为以下两方面的工作点:

- 我们在第三章提出一种 Happens-Before 弱内存模型，OHMM，它是基于操作语义的内存模型。这个模型的定义方式避免了 HMM 中可能出现的 out-of-thin-air 的行为。同时，我们使用了一种称之为重放的机制来允许程序中某些符合一定条件的指令在程序执行过程中能够多次执行，来模拟现实世界中的编译器和处理器优化中常见的投机执行。总的来说，我们的模型对于无锁程序的约束会比 Java 内存模型 (JMM) 更加弱一些，因此我们将会允许更多的编译器优化算法在我们的模型上能够使用。同时，在 OHMM 上，程序行为在直观上会比 JMM 更加自然。许多在 JMM 上可能出现但是明显违反直观认识的程序，在我们的模型上就不再合法。我们还研究了常见的简单程序变换在 OHMM 下的正确性, 这些程序变换有些在 JMM 中是正确的但是我们的模型却能很好的支持它们。我们希望可以 OHMM 可以成为可供类 Java 语言选择的一种新内存模型。同时，我们在第四章中论证了作为工业界广泛使用的弱内存模型 TSO, 其允许的程序行为是 OHMM 的行为子集。我们只需要往 OHMM 上添加额外的约束，就能够模拟出 TSO 的行为。因此，OHMM 具有良好的适配性。

- 我们在第五章中，基于 TSO 弱内存模型之上，设计了一种能够描述 TSO 中线程写缓存状态的程序逻辑，从而支持了验证 TSO 中的弱行为。借助这套逻辑系统，能让我们验证程序在 TSO 弱内存模型上的正确性。我们的程序逻辑是基于 LRG，因此我们同样支持 TSO 的局部推理，这可以使我们逻辑的表达能力大大加强并且能够验证一些很有代表性的程序在 TSO 上的行为。具体来说，我们证明了 Concurrent GCD 算法以及 Optimzed Implementation of Locks 算法在 TSO 模型上的正确性，即我们证明出这两个程序无需加入额外的同步指令，就能够像其在顺序一致性模型下，在 TSO 上得到正确的运行结果。同时我们还验证了 Peterson's Lock 算法和 Simpson's Four Slot 算法，这两个算法需要往程序中加入 **mfence** 才能够在 TSO 下正确运行，保证程序行为和程序员所设想的程序语义一致。我们同样也使用我们的逻辑系统验证了经过修改后的这两个程序在 TSO 弱内存模型上的正确性。

进一步的工作

- **支持更多语言特性。** OHMM 所采用的语言是一个 toy language, 缺少许多现代并发语言的特性比如线程创建, 线程等待, 常量域初始化, 对象初始化等等。因此我们的模型还是在这方面还是比较初步的, 要完全取代 JMM 或者成为 Java 可选的内存模型还为时尚早。因此我们希望我们能往 OHMM 中加入更多的语言特性。

- **如何进行实现。** 我们想在接下来的工作中对 OHMM 如何在现有的处理器架构上能够有效的实现进行探讨和证明。这方面目前只是一个初步的设想, 粗略的说, 由于在我们模型中,volatile 变量的读/写操作行为和 C11 模型中的 Load Require/Store Release 的 atomic 操作类似, 因此在 Power 架构上我们可以使用 “ld; cmp; bc; isync”/ “lwsync; st” 来对 volatile 变量的读/写进行实现; 而在 X86-TSO 上, 我们可以使用 “ld; mfence”/ “mfence; st” 来实现, 但是这可能会比在我们模型上的约束要强一些。

- **逻辑的辅助变量。** 第五章提出的 LRGTSO 可以支持对写操作添加辅助变量表示。即把对内存的写操作和辅助变量的写操作作为一组原子操作，同时放入缓存，在将来某个时候从缓存队列中出队并原子地同时写入内存中。然而，我们目前还没办法对内存的读操作进行同样的辅助变量标记。因为在 TSO 模型中，读操作是不需要缓存的，所以辅助变量的值并没法用于判断它所标记的读操作是否完成与否。这可能会导致我们模型对有些细粒度并发算法的证明思路需要绕一个弯。比如下面这个例子，我们想去证明在 TSO 模型下，当程序 (a) 执行完毕后, r_1 和 r_2 不可能同时为 0。如果是按照在顺序一致性模型下使用 Rely/Guarantee 推理的思路，我们应当如 (b) 中红色标示那样加入辅助变量 a 和 b。但是由于我们的逻辑并不支持对读操作添加辅助变量，所以如 (c) 所示我们在读操作和写操作之间添加辅助变量实现，而证明的思路则是我们将会有这样的不变式子在整个程序运行中成立:$(b = 1 \wedge r_2 = 0 \Rightarrow a = 0) \wedge (c = 1 \wedge r_1 = 0 \Rightarrow a = 1)$。因此我们需要去研究如何对读变量添加辅助变量标示。

  例 6.0.1. *Initially* $x = y = 0$.

$$
\begin{array}{l||l}
x := 1; & y := 1; \\
r_1 := y; & r_2 := x; \\
\textbf{mfence}\ ; & \textbf{mfence}\ ;
\end{array}
\qquad (a)
$$

$$
\begin{array}{l||l}
x := 1; & y := 1; \\
< r_1 := y, & < r_2 := x, \\
\quad \mathit{if}\ (r_1 = 0) a := 1 >; & \quad \mathit{if}\ (r_2 = 0) b := 1 >; \\
\textbf{mfence}\ ; & \textbf{mfence}\ ;
\end{array}
\qquad (b)
$$

$$
\begin{array}{l||l}
x := 1; & y := 1; \\
< a := 0, b := 1 >; & < a = 1, c = 1 >; \\
r_1 := y; & r_2 := x; \\
\textbf{mfence}\ ; & \textbf{mfence}\ ;
\end{array}
\qquad (c)
$$

参考文献

[1] Adve S V, Gharachorloo K. Shared Memory Consistency Models: A Tutorial. IEEE Computer, 1996, 29(12):66–76.

[2] Boehm H J. Threads cannot be implemented as a library. Proceedings of PLDI 2005. ACM, 2005. 261–268.

[3] Lamport L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput., 1979, 28(9):690–691.

[4] EDijkstra. Cooperating sequential processes. Proceedings of Programming Languages, London: Academic Press, 1968. 43–112.

[5] Vafeiadis V. Concurrent Separation Logic and Operational Semantics. Electron. Notes Theor. Comput. Sci., 2011, 276:335–351.

[6] Jones C B. Tentative Steps Toward a Development Method for Interfering Programs. ACM Trans. Program. Lang. Syst., 1983, 5(4):596–619.

[7] Feng X. Local rely-guarantee reasoning. Proceedings of POPL, 2009. 315–327.

[8] Vafeiadis V, Parkinson M J. A Marriage of Rely/Guarantee and Separation Logic. Proceedings of CONCUR, 2007. 256–271.

[9] Feng X, Ferreira R, Shao Z. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. Proceedings of ESOP, 2007. 173–188.

[10] Atig M F, Bouajjani A, Burckhardt S, et al. On the Verification Problem for Weak Memory Models. Proceedings of Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, Inc., 2010.

[11] Gosling J, Joy B, Steele G, et al. The Java language specifications, 2011. http://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[12] Manson J, Pugh W, Adve S V. The Java memory model. Proceedings of POPL 2005. ACM, 2005. 378–391.

[13] Cenciarelli P, Knapp A, Sibilio E. The Java Memory Model: Operationally, Denotationally, Axiomatically. Proceedings of ESOP 2007, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007. 331–346.

[14] Aspinall D, Ševčík J. Java Memory Model Examples: Good, Bad and Ugly. Proceedings of VAMP 2007, 2007.

[15] Ridge T. A Rely-Guarantee Proof System for x86-TSO. Proceedings of VSTTE10.

[16] Vafeiadis V, Narayan C. Relaxed separation logic: a program logic for C11 concurrency. Proceedings of OOPSLA, 2013. 867–884.

[17] Sevcik J, Vafeiadis V, Zappa Nardelli F, et al. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. Journal of the ACM, 2013, 60(3):22.

[18] Demange D, Laporte V, Zhao L, et al. Plan B: A Buffered Memory Model for Java. Proceedings of POPL, 2013. 329–342.

[19] Owens S, Sarkar S, Sewell P. A better x86 memory model: x86-TSO. Proceedings of TPHOLs, 2009. 391–407.

[20] Manson J. The Java Memory Model[D]. University of Maryland, College Park, 2004.

[21] Huisman M, Petri G. The Java memory model: a formal explanation. Proceedings of VAMP 2007, 2007.

[22] Boehm H J, Adve S. Foundations of The C++ concurrency Memory Model. PLDI, 2008..

[23] McKenney P E, Silvera R. Example POWER implementation for C/C++ memory model, 2011. http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2011.03.04a.html.

[24] Batty M, Dodds M, Gotsman A. Library Abstraction for C/C++ Concurrency. Proceedings of Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2013. 235–248.

[25] Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM, 1978, 21(7):558–565.

[26] Aspinall D, Ševčík J. Formalising Java's Data Race Free Guarantee. Proceedings of TPHOLs 2007, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007. 22–37.

[27] Hoare C A R. An axiomatic basis for computer programming, 1969.

[28] Reynolds J C. Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA: IEEE Computer Society, 2002. 55–74.

[29] Ishtiaq S S, O'Hearn P W. BI As an Assertion Language for Mutable Data Structures. Proceedings of POPL, 2001. 14–26.

[30] Pugh W. Java Memory Model Causality Test Cases, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html.

[31] Zhang Y, Feng X. An Interpreter and Coq Implementation of OHMM. http://staff.ustc.edu.cn/~xyfeng/research/publications/OHMM.html.

[32] Ševčík J, Aspinall D. On Validity of Program Transformations in the Java Memory Model. Proceedings of ECOOP 2008, volume 5142 of *Lecture Notes in Computer Science*. Springer, 2008. 27–51.

[33] Vafeiadis V, Balabonski T, Chakraborty S, et al. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. Proceedings of Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, 2015. 209–220.

[34] Yang Y, Gopalakrishnan G, Lindstrom G. Specifying Java thread semantics using a uniform memory model. Proceedings of Proc. 2002 Joint ACM-ISCOPE Conf. on Java Grande 2002. ACM, 2002. 192–201.

[35] Lochbihler A. Java and the Java Memory Model - A Unified, Machine-Checked Formalisation. Proceedings of ESOP 2012, volume 7211 of *Lecture Notes in Computer Science*. Springer, 2012. 497–517.

[36] Saraswat V A, Jagadeesan R, Michael M M, et al. A theory of memory models. Proceedings of PPoPP 2007. ACM, 2007. 161–172.

[37] Jagadeesan R, Pitcher C, Riely J. Generative Operational Semantics for Relaxed Memory Models. Proceedings of ESOP 2010, volume 6012 of *Lecture Notes in Computer Science*. Springer, 2010. 307–326.

[38] Sarkar S, Sewell P, Alglave J, et al. Understanding POWER multiprocessors. Proceedings of Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA: ACM, 2011. 175–186.

[39] Boudol G, Petri G. Relaxed memory models: an operational approach. Proceedings of POPL 2009. ACM, 2009. 392–403.

[40] Herlihy M, Shavit N. The Art of Multiprocessor Programming. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[41] Wang S, Wang X. Proving Simpson's Four-Slot Algorithm Using Ownership Transfer. In: Aderhold M, Autexier S, Mantel H, (eds.). Proceedings of VERIFY-2010, volume 3 of *EPiC Series*. EasyChair, 2012. 126–140.

[42] Wehrman I, Berdine J. A Proposal for Weak-Memory Local Reasoning. Proceedings of LOLA, 2011.

[43] Turon A, Vafeiadis V, Dreyer D. GPS: navigating weak memory with ghosts, protocols, and separation. Proceedings of 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. 691–707.

[44] Burckhardt S, Alur R, Martin M M K. CheckFence: checking consistency of concurrent data types on relaxed memory models. Proceedings of PLDI, 2007. 12–21.

[45] Kuperstein M, Vechev M T, Yahav E. Automatic inference of memory fences. Proceedings of FMCAD, 2010. 111–119.

[46] Alglave J, Maranget L. Stability in Weak Memory Models. Proceedings of CAV11.

[47] Vafeiadis V, Nardelli F Z. Verifying Fence Elimination Optimisations. Proceedings of SAS, 2011. 146–162.

附录 A　Simulation 引理证明

为了证明引理的方便，我们把 OHMM 的操作语义加上了标签，如图 A.3所示。我们在本附录中对定理 3.4.1进行证明：

证明. 由于 $(P', \sigma') \sim_{P^0}^{\sigma^0 \Sigma^0} (P', \Sigma')$, 所以我们有对于任意的弱执行:
$(P^0, \sigma^0) \twoheadrightarrow^* \xrightarrow{lab_0} (P^1, \sigma^1) \cdots \twoheadrightarrow^* \xrightarrow{lab_{n-1}} (P^n, \sigma^n) = (P', \sigma')$
和
$(P^0, \Sigma^0) \xrightarrow{lab_0} (P^1, \Sigma^1) \ldots \xrightarrow{lab_{n-1}} (P^n, \Sigma^n) = (P', \Sigma')$ 成立，并且对于 $j \in [0, n]$，我们都有 $(P^j, \sigma^j) \approx (P^j, \Sigma^j)$
假设 $(P', \sigma') \twoheadrightarrow^* (P', \overline{\sigma}) \xrightarrow{lab} (P'', \sigma'')$。
我们需要去证明存在 Σ'' 使得 $(P', \Sigma') \xrightarrow{lab} (P'', \Sigma'')$ 以及 $(P'', \sigma'') \approx (P'', \Sigma'')$。为了证明方便，我们假设在 σ^0 和 Σ^0 中时间等于 0。通过展开 $\approx$ 的定义, 我们知道 $(P', \overline{\sigma}) \approx (P', \Sigma')$. 我们对 *lab* 的值进行分情况讨论:

1. $lab = (\mathbf{R}\ x, i, n)$。显然 *lab* 是一个从 (P', Σ') 出发可执行的标签。假设 $(P', \Sigma') \xrightarrow{lab} (P'', \Sigma'')$. 同样，假设 $e = \langle ts, r := x \rangle$，其中 e 是在这一步弱执行中被放入缓存中的事件。

   (a) 对于从 (P'', σ'') 出发的清空缓存的转换，并且没有重放事件 e，我们有:
   $$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$
   其中，$lab_i = (\mathbf{BufA}\ (\mathbf{Exe}\ e), i, t)$ 并且 $t = ts.t$。
   假设 $lab_j = (\mathbf{BufA}\ ba, j, t)$，我们可以对 lab_j 进行讨论。

   - 若 $i = j$。根据依赖性的定义，我们知道在这两个缓存动作之间，并不存在任何依赖性。因此我们可以在状态变换中交换这两个标签发生的位置并且达到相同的状态 σ_j。
   - 若 $i \neq j \wedge ba \notin \{s \mid s = \mathbf{Exe}\ \_ \vee s = \mathbf{Rep}\ \_\}$。在这种情况中，$lab_j$ 只能是清空线程 j 的混存或者删除线程 j 发出的某个写操作时间。这两者都不能影响其他线程事件的执行结果。因此我们可以知道，我们同样可以交换这两个标签所发生的位置，并且达到相同的状态 σ_j。
   - 若 $i \neq j \wedge ba = \mathbf{Exe}\ e'$。在这种情况下，如果 e' 不是对变量 x 的写操作，那么我们仍然可以通过交换 lab_i 和 lab_j 的位置来达到同一个状态 σ_j。如果 e' 是对 x 变量的写操作，这意味着 e' 是在 e 之后执行的写操作。假设 e' 的时间是 t'。根据图 A.3和图 A.1所示的操作语义，我们知道在弱执行和强执行中都存在 $lab' = (Wx, j, t')$。因为 (P^0, Σ^0) 没有数据竞争，我们知道存在 $Rel\text{-}Acq_{lab_p}^{lab_q}$ 并且有 $t' < lab_p.t < lab_q.t < lab.t$, $lab_p.tid = j$ 以及 $lab_q.tid = i$. 所以我们知道 e 一定要在 e' 之后执行，这和我们前面所说的矛盾，因此这种情况不可能发生。
   - 若 $i \neq j \wedge ba = \mathbf{Rep}\ e'$。这种情况和上面类似，如果 e' 是一个对除 x 以外的变量的写操作，我们可以交换两个标签的位置。否则，我们会通过得出矛盾来推知 e' 不可能是对 x 的写操作。

所以我们现在能有，无论 lab_j 是什么，我们总可以把它和 lab_i 在缓存清空转换中交换位置以后，仍然达到相同的末状态。即：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_i') \xrightarrow{lab_i} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

. 我们可以如上所述把 lab_i 和转换中位于它后面的标签交换若干次，直到我们最后把标签 $(\textbf{BufA}\ (\textbf{Exe}\ e), ts)$ 换到缓存清空转换中的最后一个标签，我们仍然可以达到相同的末状态，因此我们有：

$$(P'', \sigma'') \twoheadrightarrow^* (P'', \sigma_h) \xrightarrow{lab_i} (P'', \sigma_m) \qquad (*)$$

并且我们通过图 A.2中的 **ISSUE-R** 规则知道 σ'' 和 $\overline{\sigma}$ 除了 $\sigma''.B = \overline{\sigma}.B \cup \{e\}$ 以外其他都是相同的。因此对于任意的如 (*) 的缓存清空转换，我们都可以有对应的每一步的标签都一样的从 $(P', \overline{\sigma})$ 出发的缓存清空转换：

$$(P', \overline{\sigma}) \Downarrow (P', \sigma_h')$$

使得 $\sigma_h.tq = \sigma_h'.tq$, $\sigma_h.m = \sigma_h'.m$, $\sigma_h.t = \sigma_h'.t + 1$, $\sigma_h.L = \sigma_h'.L$ 和 $\sigma_h.b = \sigma_h'.b \uplus e$。我们知道 $(P', \overline{\sigma}) \approx (P', \Sigma')$ 并且 $(P', \overline{\sigma}) \Downarrow (P', \sigma_h')$，所以我们有 $\sigma_h' \overset{\text{MRLT}}{=\!=} \Sigma'$。我们还知道 $\sigma_m.tq = \sigma_h.tq\{i \rightsquigarrow (rf', \emptyset)\}$, $rf' = \sigma_h.tq(i).rf\{r \rightsquigarrow V\}$, $V = Value(\sigma_h, x)$, $\sigma_m.m = \sigma_h.m$, $\sigma_m.t = \sigma_h.t + 1$, $\sigma_m.L = \sigma_h.L$ 以及 $\sigma_m.b = \emptyset$。另外一方面，我们有 $\Sigma'.L = \Sigma''.L$, $\Sigma'.m = \Sigma'.m$, $\Sigma'.t = \Sigma''.t - 1$ 和 $\Sigma''.TQ = \Sigma'.TQ\{i \rightsquigarrow rf'\}$。因此通过展开 $\approx$ 的定义，我们知道 $(P'', \sigma_m) \approx (P'', \Sigma'')$。

(b) 对于从 (P'', σ'') 出发的缓存清空转换，并且重放了事件 e。我们有：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 以及 $t = ts.t$。假设 lab_i 是最后一个 e 的重放标签。如同我们上面讨论的那样，在标签 lab_i 和 lab_j 的转换中，并不存在对 x 的写操作，否则就发生了数据竞争，和我们的前提矛盾。因此抽象机在 lab_i 和 lab_j 中读到的值是一样的。因此我们可以简单地把这个重放标签从转换中去掉，并且达到相同的末状态 σ_m。同样的，我们可以如此这般把之前所有的对 lab_i 的重放标签都去掉并且达到同一个末状态 σ_m。所以我们最终会得到一个没有重放事件 e 的转换，然后我们就能用 Case 1a相同的方法来得出结论。

2. $lab = (\text{exp}\ r, i, n)$. 如同我们之前分析的那样，$lab$ 是从 (P', Σ') 开始的可执行动作。假设 $(P', \Sigma') \xrightarrow{lab} (P'', \Sigma'')$ 以及 $e = \langle ts, r := E \rangle$，其中 e 是在这一步弱执行中放入缓存的事件。

(a) 对于所有的 (P'', σ'') 出发的清空缓存转换，并且没有重放事件 e, 我们有：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 并且 $t = n + 1$。由于 lab_i 并没有访问内存，因此我们可以简单地将其和后续的标签进行交换位置，直到它成为这个转换中的最后一个标签，同时，我们还能知道，末状态是一样的。所以我们有：

$$(P'', \sigma'') \twoheadrightarrow^* (P'', \sigma_h) \xrightarrow{lab_i} (P'', \sigma_m) \qquad (**)$$

并且我们从知道图 A.2中的 **ISSUE-EXP** 规则知道，σ'' 和 $\overline{\sigma}$ 相同，除了 $\sigma''.B = \overline{\sigma}.B \cup \{e\}$。因此对于任意的如 (**) 的转换，我们可以有一个相应的，从 $(P', \overline{\sigma})$ 开始的每一步标签都一样的内存清空转换：

$$(P', \overline{\sigma}) \Downarrow (P', \sigma'_h)$$

使得 $\sigma_h.tq = \sigma'_h.tq$, $\sigma_h.m = \sigma'_h.m$, $\sigma_h.t = \sigma'_h.t + 1$, $\sigma_h.L = \sigma'_h.L$ 以及 $\sigma_h.b = \sigma'_h.b \uplus e$. 我们知道 $(P', \overline{\sigma}) \approx (P', \Sigma')$, 并且 $(P', \overline{\sigma}) \Downarrow (P', \sigma'_h)$, 因此我们有 $\sigma'_h \overset{\text{MRLT}}{=\!=\!=} \Sigma'$. 我们还直到 $\sigma_m.tq = \sigma_h.tq\{i \rightsquigarrow (rf', \emptyset)\}$, $rf' = \sigma_h.tq(i).rf\{r \rightsquigarrow V\}$, $V = [\![E]\!]_{\sigma_h.tq(i).rf}$, $\sigma_m.m = \sigma_h.m$, $\sigma_m.t = \sigma_h.t + 1$, $\sigma_m.L = \sigma_h.L$ 并且 $\sigma_m.b = \emptyset$. 另外一方面由于我们有 $\Sigma'.L = \Sigma''.L$, $\Sigma'.m = \Sigma'.m$, $\Sigma'.t = \Sigma''.t - 1$, $\Sigma''.TQ = \Sigma'.TQ\{i \rightsquigarrow rf'\}$，因此我们能得出 $(P'', \sigma_m) \approx (P'', \Sigma'')$.

item 对于所有的 (P'', σ'') 出发的清空缓存转换，并且重放事件 e，我们有:

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 并且 $t = n + 1$. 假设 lab_i 是第一个重放事件 e 的标签。
若 lab_i 和 lab_j 读到不同的值，我们知道在转换中一定存在 e' 使得：

$$\begin{aligned}&(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_k} (P'', \sigma_k) \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i)\\ &\twoheadrightarrow^* \xrightarrow{lab_h} (P'', \sigma_h) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)\end{aligned}$$

其中 $lab_k = (\textbf{BufA}\ (\textbf{Rep}\ e'), i, t)$，$lab_h = (\textbf{BufA}\ (\textbf{Exe}\ e'), i, t)$, $UpdR(e'.\iota) \subseteq UseR(e.\iota)$ 并且 **Rep** e' 和 **Exe** e' 读到不同的值. 若 $e'.\iota$ 是一条纯指令，即它没有访问内存，那么我们用我们找到 e 的相同方法，继续在转换中寻找某个 e''，直到我们找到 e_r 访问了内存并且在 **Rep** e_r 和 **Exe** e_r 时读到不同的值。如果不存在这么一个 e_r, 则根据我们的操作语义我们知道所有我们找到的事件都是纯指令，因此它们的值不会被修改，无论它们被重放多少次。如果我们找到了 e_r, 就如我们在 Case 1a中分析的哪样，我们知道不管事件 e_r 被重放多少次，它总是读到相同的值。因此我们就得到了一个矛盾。所以我们知道 **Rep** e and **Exe** e 读到相同的值。所以我们可以把所有的对 e 的重放标签都从转换中去掉，并达到相同的末状态 σ_m。然后接下来的这个 Case 就变成了 Case 2a的情况。我们可以像上面一样分析得到结论。

3. $lab = (\textbf{W}\ x, i, t)$. 同样我们可以分为两种情况讨论：

   (a) 假设在这一步被放进缓存的事件是 $e = \langle ts, x := n_1 \rangle$.

      i. 对于所有的 (P'', σ'') 出发的清空缓存转换，并且没有重放事件 e，我们有：

      $$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* (P'', \sigma_m)$$

      其中 $lab_i = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 如同我们之前做的哪样，我们可以很容易将 $(\textbf{BufA}\ (\textbf{Exe}\ e), ts)$ 移动到转换中的最后一个标签，并且达到相同的末状态。所以我们有

      $$(P'', \sigma'') \twoheadrightarrow^* (P'', \sigma_h) \xrightarrow{lab_i} (P'', \sigma_m) \qquad (***)$$

并且我们从知道图 A.2中的 **ISSUE-W** 规则知道，σ'' 和 $\overline{\sigma}$ 相同，除 $\sigma''.B = \overline{\sigma}.B \cup \{e\}$。因此对于任意的如 (***) 的缓存转换序列，我们都可以有从 $(P', \overline{\sigma})$ 开始的相同的缓存转换序列

$$(P', \overline{\sigma}) \Downarrow (P', \sigma'_h)$$

使得 $\sigma_h.tq = \sigma'_h.tq$, $\sigma_h.m = \sigma'_h.m$, $\sigma_h.t = \sigma'_h.t + 1$, $\sigma_h.L = \sigma'_h.L$ 和 $\sigma_h.b = \sigma'_h.b \uplus e$。我们知道 $(P', \overline{\sigma}) \approx (P', \Sigma')$, 并且 $(P', \overline{\sigma}) \Downarrow (P', \sigma'_h)$，所以我们有 $\sigma'_h \overset{MRLT}{=\!=} \Sigma'$。我们还知道 $\sigma_m.tq = \sigma_h.tq$, $\sigma_m.m = Add(\sigma_h.m, x, ts, n_1)$, $\sigma_m.t = \sigma_h.t + 1$, $\sigma_m.L = \sigma_h.L$ 和 $\sigma_m.b = \emptyset$。另外一方面，我们知道 $\Sigma'.L = \Sigma''.L, \Sigma'.m = \Sigma'.m\{x \rightsquigarrow n_1\}, \Sigma'.t = \Sigma''.t - 1, \Sigma''.TQ = \Sigma'.TQ\{i \rightsquigarrow rf'\}$。所以根据 *Value* 的定义, 我们有 $Value(\sigma_m, x) = n_1$, 否则就会产生数据竞争，和我们的假设矛盾。因此我们可以有 $(P'', \sigma_m) \approx (P'', \Sigma'')$。

ii. 对于所有的 (P'', σ'') 出发的清空缓存转换，并且重放事件 e，如果 e 没有被从转换中删除，那么我们有：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 以及 $t = n + 1$。假设 lab_i 是缓存清空序列中 e 的最后一个重放标签。由于 n_1 是常数，所以在 **Rep** e 和 **Exe** e 中所写入的值都是一样的，因此 **Exe** e 不会改变 **Rep** e 写入历史记录的值。因此我们可以把序列中所有 e 的重放标签都去掉，并且达到相同的状态 σ^m。所以我们可以如在 Case 3(a)i中分析的那样得出结论。
如果 e 从转换中删除了，则我们有

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 以及 $t = n+1$. 假设 lab_i 是缓存清空序列中 e 的最后一个重放标签。根据操作语义我们知道在 lab_i 和 lab_j 之间存在 lab' 使得其读取 x 的值。然而，由于 (P^0, Σ^0) 没有数据竞争，所以这种情况不会发生。

(b) $e = \langle ts, x := r \rangle$. 如同我们之前做的那样，我们可以把这种情况细分为三种：

i. 对于所有的 (P'', σ'') 出发的清空缓存转换，并且没有重放事件 e。对于这种情况的证明，和 Case 3(a)i一样。

ii. 对于所有的 (P'', σ'') 出发的清空缓存转换，并且重放事件 e，如果 e 没有被从转换中删除，那么我们有：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 并且 $t = n + 1$。假设 lab_i 是缓存清空序列中 e 的最后一个重放标签。如果 r 的值在 **Exe** e 和 **Rep** e 之间是不同的。我们可以用 Case 2a中相同方法来导出矛盾。所以 r 的值一定相同，故如同上面你分析的一样，我们可以把所有的重放事件都删去，然后得到 Case 3(b)i中的情况。

iii. 对于所有的 (P'', σ'') 出发的清空缓存转换，并且重放事件 e，如果 e 被从转换中删除，那么我们有：

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* \xrightarrow{lab_j} (P'', \sigma_j) \twoheadrightarrow^* (P'', \sigma_m)$$

其中 $lab_i = (\textbf{BufA}\ (\textbf{Rep}\ e), i, t)$, $lab_j = (\textbf{BufA}\ (\textbf{Exe}\ e), i, t)$ 以及 $t = n+1$. 假设 lab_i 是缓存清空序列中 e 的最后一个重放标签。根据操作语义我们知道在 lab_i 和 lab_j 之间存在 lab' 使得其读取 x 的值。然而，由于 (P^0, Σ^0) 没有数据竞争，所以这种情况不会发生。

4. $lab = (\textbf{cmp}\ r_k, i, n)$. 在这种情况下，我们知道 σ'' 和 $\overline{\sigma}$ 相同, 并且 $\Sigma'' = \Sigma'$。显然，我们有 $(P'', \sigma'') \approx (P'', \Sigma'')$. 所以我们只需证明线程 i 中的 r_k 的值在 $\overline{\sigma}$ 和 Σ' 中是一样的。由于 $(P', \overline{\sigma}) \approx (P', \Sigma')$, 所以存在 σ_h 使得 $(P', \overline{\sigma}) \Downarrow (P', \sigma_h)$。因此我们有 $\sigma_h.tq(i).rf = \Sigma'.tq(i).rf$. 由于 lab 是从 $(P', \overline{\sigma})$ 出发的可执行标签，意味着在缓存中没有任何的事件可以写入寄存器 r_k。因此我们有 $\overline{\sigma}.tq(i).rf(r_k) = \sigma_h.tq(i).rf(r_k)$，最后我们得到 $\overline{\sigma}.tq(i).rf(r_k) = \Sigma'.tq(i).rf(r_k)$。

5. $lab = (\textbf{acq}\ l, i, n)$. 由于 **lock** 在强执行和弱执行中都是立即执行的语句，所以我们可以直接得出结论。

6. $lab = (\textbf{ld}\ v, i, n)$ 或者 $lab = (\mathsf{VRel}\ v, i, n)$. 如我们之前做的那样，我们有

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* (P'', \sigma_m)$$

由于在缓存清空转换序列中不存在任何的标签是写或者读 volatile 变量 v（volatile 事件的执行顺序必须和 synchronization order 一致）。因此我们可以把 lab_i 移动到序列的最后，并且仍然可以达到相同的状态。因此我们很容易得出 $\sigma_m \overset{MRLT}{=\!=} \Sigma''$。所以我们能够证明 $(P'', \sigma'') \approx (P'', \Sigma'')$。

7. $lab = (\textbf{rel}\ l, i, n)$. 我们有

$$(P'', \sigma'') \twoheadrightarrow^* \xrightarrow{lab_i} (P'', \sigma_i) \twoheadrightarrow^* (P'', \sigma_m)$$

由于在缓存清空转换序列中不存在任何的标签是释放 l 的事件（lock 和 unlock 事件的执行顺序必须和 synchronization order 一致）。因此我们可以把 lab_i 移动到序列的最后，并且仍然可以达到相同的状态。因此我们很容易得出 $\sigma_m \overset{MRLT}{=\!=} \Sigma''$。所以我们能够证明 $(P'', \sigma'') \approx (P'', \Sigma'')$。

□

$$\frac{tq(\mathtt{i})(r) = \mathtt{n} \quad C = \mathtt{x} := r; C' \quad \mathtt{m}' = \mathtt{m}\{\mathtt{x} \rightsquigarrow \mathtt{n}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{W}\,\mathtt{x},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m', \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{m(\mathtt{x}) = \mathtt{n} \quad tq(\mathtt{i}) = \mathtt{rf} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\} \quad C = r := \mathtt{x}; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{R}\,\ell,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{C = r := E; C' \quad tq(\mathtt{i}) = rf \quad [\![E]\!]_{rf} = \mathtt{n} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{exp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = \mathtt{n} \quad C = \mathtt{v} := r; C' \quad \mathtt{m}' = \mathtt{m}\{\mathtt{v} \rightsquigarrow \mathtt{n}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{st}\,\mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m', \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{m(\mathtt{v}) = \mathtt{n} \quad tq(\mathtt{i}) = \mathtt{rf} \quad rf' = rf\{r \rightsquigarrow \mathtt{n}\} \quad tq' = tq\{\mathtt{i} \rightsquigarrow rf'\} \quad C = r := \mathtt{v}; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{ld}\,\mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq', m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = 0 \quad C = \mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_2; C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) \neq 0 \quad C = \mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_1; C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) = 0 \quad C = \mathbf{while}\ r\ \mathbf{do}\ C_1; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{tq(\mathtt{i})(r) \neq 0 \quad C = \mathbf{while}\ r\ \mathbf{do}\ C_1; C'}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\,r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C_1; C], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}\rangle)}$$

$$\frac{C = \mathbf{lock}\ \mathtt{l}; C' \quad \mathtt{l} \notin \mathsf{dom}(\mathtt{L}) \quad \mathtt{L}' = \mathtt{L}\{\mathtt{l} \rightsquigarrow \mathtt{i}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{acq}\,\mathtt{l},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}'\rangle)}$$

$$\frac{C = \mathbf{unlock}\ \mathtt{l}; C' \quad \mathtt{L}(\mathtt{l}) = \mathtt{i} \quad \mathtt{L}' = \mathtt{L} \setminus \{\mathtt{l}\}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{rel}\,\mathtt{l},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{t}{+}1, \mathtt{L}'\rangle)}$$

图 A.1: 带标签的交叉语义

$$\frac{C = \iota; C' \quad \iota = \mathtt{x} := \mathtt{n} \quad e = \langle\langle \mathtt{i}, \mathtt{t}\rangle, \iota\rangle \quad \mathtt{b}' = \mathtt{b} \cup \{e\}}{(\mathbb{P}[\mathtt{i}.C], (tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L})) \xrightarrow{(\mathbf{W}\ \mathtt{x},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], tq, m, \mathtt{b}', \mathtt{t}{+}1, \mathtt{L})} \text{(issue-W)}$$

$$\frac{C = \iota; C' \quad \iota = \mathtt{r} := \mathtt{x} \quad e = \langle\langle \mathtt{i}, \mathtt{t}\rangle, \iota\rangle \quad \mathtt{b}' = \mathtt{b} \cup \{e\}}{(\mathbb{P}[\mathtt{i}.C], (tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L})) \xrightarrow{(\mathbf{R}\ \mathtt{x},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], tq, m, \mathtt{b}', \mathtt{t}{+}1, \mathtt{L})} \text{(issue-R)}$$

$$\frac{C = \iota; C' \quad \iota = \mathtt{r} := \mathtt{v} \quad e = \langle\langle \mathtt{i}, \mathtt{t}\rangle, \iota\rangle \quad \mathtt{b}' = \mathtt{b} \cup \{e\}}{(\mathbb{P}[\mathtt{i}.C], (tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L})) \xrightarrow{(\mathbf{ld}\ \mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], tq, m, \mathtt{b}', \mathtt{t}{+}1, \mathtt{L})} \text{(issue-LD)}$$

$$\frac{C = \iota; C' \quad \iota = \mathtt{v} := \mathtt{n} \quad e = \langle\langle \mathtt{i}, \mathtt{t}\rangle, \iota\rangle \quad \mathtt{b}' = \mathtt{b} \cup \{e\}}{(\mathbb{P}[\mathtt{i}.C], (tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L})) \xrightarrow{(\mathbf{st}\ \mathtt{v},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], tq, m, \mathtt{b}', \mathtt{t}{+}1, \mathtt{L})} \text{(issue-ST)}$$

$$\frac{C = \iota; C' \quad \iota = \mathtt{r} := \mathtt{E} \quad e = \langle\langle \mathtt{i}, \mathtt{t}\rangle, \iota\rangle \quad \mathtt{b}' = \mathtt{b} \cup \{e\}}{(\mathbb{P}[\mathtt{i}.C], (tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L})) \xrightarrow{(\mathbf{exp}\ \mathtt{r},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], tq, m, \mathtt{b}', \mathtt{t}{+}1, \mathtt{L})} \text{(issue-exp)}$$

$$\frac{\begin{array}{c} C = \mathbf{lock}\ \mathtt{l}; C' \quad \mathtt{l} \notin \mathsf{dom}(\mathtt{L}) \\ \mathtt{L}' = \mathtt{L}\{\mathtt{l} \rightsquigarrow \mathtt{i}\} \quad m' = \mathit{AddSyn}(m, \langle\langle \mathtt{i}, \mathtt{t}\rangle, \mathbf{acq}, \mathtt{l}\rangle) \end{array}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{acq}\ \mathtt{l},\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m', \mathtt{b}, \mathtt{t}{+}1, \mathtt{L}'\rangle)} \text{(lk-Acq)}$$

$$\frac{\begin{array}{c} C = (\mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2); C' \quad C'' = C_1; C' \\ \mathit{readyR}(\langle \mathtt{i}, \mathtt{t}\rangle, r, \mathtt{b})tq(\mathtt{i}).rf(r) \neq 0 \end{array}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\ r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C''], \langle tq, m, \mathtt{b}, \mathtt{t}{+}1, \mathtt{L}\rangle)} \text{(if-t)}$$

$$\frac{\begin{array}{c} C = (\mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2); C' \quad C'' = C_2; C' \\ \mathit{readyR}(\langle \mathtt{i}, \mathtt{t}\rangle, r, \mathtt{b}) \quad tq(\mathtt{i}).rf(r) = 0 \end{array}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\ r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C''], \langle tq, m, \mathtt{b}, \mathtt{t}{+}1, \mathtt{L}\rangle)} \text{(if-f)}$$

$$\frac{\begin{array}{c} C = (\mathbf{while}\ r\ \mathbf{do}\ C_1); C' \quad C'' = C_1; C \\ \mathit{readyR}(\langle \mathtt{i}, \mathtt{t}\rangle, r, \mathtt{b}) \quad tq(\mathtt{i}).rf(r) \neq 0 \end{array}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\ r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C''], \langle tq, m, \mathtt{b}, \mathtt{t}{+}1, \mathtt{L}\rangle)} \text{(while-t)}$$

$$\frac{\begin{array}{c} C = (\mathbf{while}\ r\ \mathbf{do}\ C_1); C' \\ \mathit{readyR}(\langle \mathtt{i}, \mathtt{t}\rangle, r, \mathtt{b}) \quad tq(\mathtt{i}).rf(r) = 0 \end{array}}{(\mathbb{P}[\mathtt{i}.C], \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{cmp}\ r,\mathtt{i},\mathtt{t})} (\mathbb{P}[\mathtt{i}.C'], \langle tq, m, \mathtt{b}, \mathtt{t}{+}1, \mathtt{L}\rangle)} \text{(while-f)}$$

$$\frac{\begin{array}{c} tq(\mathtt{i}) = (rf, rb) \quad tq' = tq\{\mathtt{i} \rightsquigarrow (rf', rb')\} \\ \langle (rf, rb), m, \mathtt{b}, \mathtt{L}\rangle \xrightarrow[\mathtt{b}\alpha]{\mathtt{i}} \langle (rf', rb'), m', \mathtt{b}', \mathtt{L}'\rangle \end{array}}{(P, \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{BufA}\ \mathtt{b}\alpha,\mathtt{i},\mathtt{t})} (P, \langle tq', m', \mathtt{b}', \mathtt{t}, \mathtt{L}'\rangle)} \text{(evt-ba)}$$

$$\frac{tq(\mathtt{i}) = (rf, rb) \quad tq' = tq\{\mathtt{i} \rightsquigarrow (rf, \emptyset)\}}{(P, \langle tq, m, \mathtt{b}, \mathtt{t}, \mathtt{L}\rangle) \xrightarrow{(\mathbf{BufA}\ \mathbf{Emp},\mathtt{i},\mathtt{t})} (P, \langle tq', m, \mathtt{b} \cup rb, \mathtt{t}, \mathtt{L}\rangle)} \text{(replay-emp)}$$

图 A.2: 带标签的 OHMM 操作语义: 从指令到事件

$$\frac{e = \langle ts, r := v\rangle \quad syn = \langle ts, \mathbf{ld}, v\rangle \quad m' = AddSyn(m, syn) \quad rf' = rf\{r \rightsquigarrow m(v)\} \quad r \notin UpdR(rb)}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf', rb), m', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(rd-v-exe)}$$

$$\frac{e = \langle ts, v := \mathsf{r}\rangle \quad syn = \langle ts, \mathbf{st}, v\rangle \quad m' = AddSyn(m, syn) \quad m'' = m'\{v \rightsquigarrow rf(r)\} \quad r \notin UpdR(rb)}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf, rb), m'', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(wt-v-exe)}$$

$$\frac{e = \langle ts, \mathbf{unlock}\ \mathsf{l}\rangle \quad \mathsf{L}(\mathsf{l}) = \mathsf{i} \quad \mathsf{L}' = \mathsf{L} \setminus \{\mathsf{l}\} \quad syn = \langle ts, \mathbf{rel}, \mathsf{l}\rangle \quad m' = AddSyn(m, syn)}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf, rb), m', \mathsf{b}\backslash\{e\}, \mathsf{L}'\rangle} \text{(unlk-exe)}$$

$$\frac{e = \langle ts, r := \mathsf{x}\rangle \quad visible(ts, \langle ts', \mathsf{n}, \_\rangle, m(\mathsf{x})) \quad ts.tid = ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathsf{n}\} \quad Replay(rb, \mathsf{e}, rb') \quad rb = rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf', rb'), m, \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(rd-self-exe)}$$

$$\frac{e = \langle ts, r := \mathsf{x}\rangle \quad visible(ts, \langle ts', \mathsf{n}, \_\rangle, m(\mathsf{x})) \quad ts.tid = ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathsf{n}\} \quad Replay(rb, \mathsf{e}, rb') \quad rb \mathsf{neq} rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Rep}\ e]{\mathsf{i}} \langle (rf', rb'), m, \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(rd-self-rep)}$$

$$\frac{e = \langle ts, r := \mathsf{x}\rangle \quad visible(ts, \langle ts', \mathsf{n}, \_\rangle, m(\mathsf{x})) \quad ts.tid \neq ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathsf{n}\} \quad m' = ModRef(m, \mathsf{x}, ts') \quad Replay(rb, \mathsf{e}, rb') \quad rb = rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf', rb'), m', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(rd-other-exe)}$$

$$\frac{e = \langle ts, r := \mathsf{x}\rangle \quad visible(ts, \langle ts', \mathsf{n}, \_\rangle, m(\mathsf{x})) \quad ts.tid \neq ts'.tid \quad rf' = rf\{r \rightsquigarrow \mathsf{n}\} \quad m' = ModRef(m, \mathsf{x}, ts') \quad Replay(rb, \mathsf{e}, rb') \quad rb \neq rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Rep}\ e]{\mathsf{i}} \langle (rf', rb'), m', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(rd-other-rep)}$$

$$\frac{e = \langle ts, \mathsf{x} := r\rangle \quad e \in \mathsf{b} \quad \langle ts, \_, \mathbf{true}\rangle \in m(\mathsf{x})}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Del}\ e]{\mathsf{i}} \langle (rf, rb), m, \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(no-wt-replay-Del)}$$

$$\frac{e = \langle ts, \mathsf{x} := r\rangle \quad rf(r) = \mathsf{n} \quad m' = Add(m, \mathsf{x}, ts, \mathsf{n}) \quad Replay(rb, \mathsf{e}, rb') \quad rb = rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf, rb'), m', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(wt-exe)}$$

$$\frac{e = \langle ts, \mathsf{x} := r\rangle \quad rf(r) = \mathsf{n} \quad m' = Add(m, \mathsf{x}, ts, \mathsf{n}) \quad Replay(rb, \mathsf{e}, rb') \quad rb \neq rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf, rb'), m', \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(wt-rep)}$$

$$\frac{e = \langle ts, \mathsf{r} := \mathsf{E}\rangle \quad [\![E]\!]_{rf} = \mathsf{n} \quad rf' = rf\{\mathsf{r} \rightsquigarrow \mathsf{n}\} \quad Replay(rb, \mathsf{e}, rb') \quad rb = rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Exe}\ e]{\mathsf{i}} \langle (rf', rb'), m, \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(pure-exe)}$$

$$\frac{e = \langle ts, \mathsf{r} := \mathsf{E}\rangle \quad [\![E]\!]_{rf} = \mathsf{n} \quad rf' = rf\{\mathsf{r} \rightsquigarrow \mathsf{n}\} \quad Replay(rb, \mathsf{e}, rb') \quad rb \neq rb'}{\langle (rf, rb), m, \mathsf{b}, \mathsf{L}\rangle \xrightarrow[\mathbf{Rep}\ e]{\mathsf{i}} \langle (rf', rb'), m, \mathsf{b}\backslash\{e\}, \mathsf{L}\rangle} \text{(pure-rep)}$$

图 A.3: 带标签的 OHMM 操作语义: 执行事件

附录 B　程序变换的正确性证明

在本节附录中，我们将给出定理3.5.1的证明。我们通过模拟关系，将目标程序和源程序的执行过程联系起来。证明过程可能看起来很有技巧并且难以理解为什么这么做，但是只要把握住证明的核心思想就是需要去构建两个不同状态转移序列中相应的状态之间的 “观测等价” 即可。因为只要我们找到这么一个关系，剩下的证明只是标准的模拟关系证明。

B.1　E-WAR 的正确性

我们为证明分为若干个部分，分别去证明每个 transformation。在这一节中，我们想取证明 E-WAR 的正确性。首先，我们先构建两个程序 (源程序和目标程序) 的以及状态之间的关系。由于状态中的缓存在执行的过程中并不能保证一定是空的，因此我们需要去定义新的观测等价，来满足模拟关系。

$$
\begin{array}{lcl}
match(e_1, e_2) & \stackrel{\text{def}}{=} & e_1.\iota = e_2.\iota \wedge e_1.ts.tid = e_2.ts.tid \\
\sigma \stackrel{b_1}{=\!=} \sigma' & \stackrel{\text{def}}{=} & \forall e \in buff(\sigma').\ \exists e' \in buff(\sigma).match(e, e') \\
 & & \wedge(\forall e_1, e_2 \in buff(\sigma').\ e_1.ts.t < e_2.ts.t \\
 & & \quad \Rightarrow \exists e_1', e_2' \in buff(\sigma).\ (match(e_1, e_1') \\
 & & \quad\ \wedge match(e_2, e_2') \wedge e_1'.ts.t < e_2'.ts.t) \\
 & & \wedge(\text{sizeof}(buff(\sigma')) = \text{sizeof}(buff(\sigma)) \\
 & & \quad \vee(\exists e_1, e_2 \in \sigma.\ Opti(e_1, e_2) \\
 & & \quad\ \wedge \neg\exists e_1' \in \sigma'.\ match(e_1, e_1') \\
 & & \quad\ \wedge \exists e_2' \in \sigma'.\ match(e_2, e_2'))) \\
Opti(e_1, e_2) & \stackrel{\text{def}}{=} & e_1 \textit{ and } e2 \textit{ are the events whoses instructions are transformated} \\
\sigma \stackrel{E\text{-}WAR}{=\!=\!=} \sigma' & \stackrel{\text{def}}{=} & \sigma \stackrel{r}{=} \sigma' \wedge \sigma \stackrel{b_1}{=\!=} \sigma' \wedge \sigma \stackrel{m}{=\!=} \sigma' \wedge \sigma \stackrel{l}{=} \sigma'
\end{array}
$$

定义 B.1.1 (The simulation relation)**.**

$$
\begin{array}{lcl}
(P, \sigma) \sim^n (P', \sigma') & \stackrel{def}{=} & (P \xrightarrow{E\text{-}WAR} P' \wedge \sigma \stackrel{E\text{-}WAR}{=\!=\!=} \sigma') \\
 & & \wedge\forall P'', \sigma''.\ (P', \sigma') \rightarrow (P'', \sigma'') \Rightarrow \\
 & & \quad \exists P''', \sigma'''.\ (P, \sigma) \rightarrow^* (P''', \sigma''') \\
 & & \quad\ \wedge (P''', \sigma''') \sim^{n-1} (P'', \sigma'') \\
(\textbf{skip}, \sigma) \sim^n (\textbf{skip}, \sigma') & \stackrel{def}{=} & \sigma \stackrel{E\text{-}WAR}{=\!=\!=} \sigma' \\
(P, \sigma) \sim^0 (P', \sigma') & \stackrel{def}{=} & P \xrightarrow{E\text{-}WAR} P' \wedge \sigma \stackrel{E\text{-}WAR}{=\!=\!=} \sigma' \\
(P, \sigma) \sim (P', \sigma') & \stackrel{def}{=} & \forall n.\ (P, \sigma) \sim^n (P', \sigma')
\end{array}
$$

引理 B.1.1. *对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E\text{-}WAR} P'$ 并且 $\sigma \stackrel{E\text{-}WAR}{=\!=\!=} \sigma'$, 同时我们假设 P' 的执行可以终止，那么我们有 $(P, \sigma) \sim (P', \sigma')$。*

证明. 我们需要证明 $\forall n.\ (P, \sigma) \sim^n (P', \sigma')$.
因此我们在 n 上做归纳假设:

- Base: $n = 0$. 我们可以直接通过定义来证明基本情况。
- Inductive: 根据归纳假设，我们有
  对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E\text{-}WAR} P'$ 并且 $\sigma \xrightarrow{E\text{-}WAR} \sigma'$ 以及 P' 是终止的, 则 $(P, \sigma) \sim^n (P', \sigma')$ 成立
  我们需要去证明:
  对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E\text{-}WAR} P'$ 并且 $\sigma \xrightarrow{E\text{-}WAR} \sigma'$ 以及 P' 是终止的, 则 $(P, \sigma) \sim^{n+1} (P', \sigma')$ 成立
  证明如下:
  对于 $\forall P'', \sigma''.\ (P', \sigma') \to (P'', \sigma'')$, 我们需要对 P'' and P' 的结构进行讨论。
  - $P'' = P'$. 意味这一步中并没有发生从指令到缓存的变换，因此根据操作语义我们有 $(P', \sigma') \xrightarrow{(\textbf{BufA}\ ba, ts')} (P', \sigma'')$, 接下来我们根据 ba 进行讨论。
    * $ba = \textbf{Exe}\ e$. 在这种情况下，我们根据 e 来进行讨论，并且假设在 σ 中匹配的事件是 is e':
      · $\exists e_1 \in \sigma,\ e_1.\iota \in \iota_n \wedge \neg\exists e_2 \in \sigma',\ match(e_1, e_2) \wedge Opti(e_1, e')$. 假设 $e_1 = \langle ts_1, x := i\rangle$ 并且 $e = \langle ts, x := ri\rangle$. 则我们有 $e' = \langle ts', x = ri\rangle$. 所以我们有 $\sigma \xrightarrow{\textbf{Exe}\ e_1} \sigma^1$。我们需要注意到当在 σ 中没有任何的依赖事件时，e_1 可以被执行。而当 σ 中 e_1 有所依赖的事件还没执行，我们令这个事件是 e_3, 所以我们知道 σ 中存在 e_3' 使得 $match(e_3, e_3')$ 并且 e 依赖于 e_3', 这将导出矛盾。所以我们让 $\sigma^1 \xrightarrow{\textbf{Exe}\ e'} \sigma'''$。现在我们仅需要证明 $\sigma''' \overset{m}{=\!=} \sigma''$。
      $$\forall i.\ visibleV(i, \sigma'''.t, h) = visibleV(i, \sigma''.t, h)$$
      其中 $h = \sigma''.m(x)$。因此我们有 $\sigma''' \xrightarrow{E\text{-}WAR} \sigma''$, 并且可以根据归纳假设得出结论。
      · 其他情况. 我们令 $\sigma \xrightarrow{\textbf{Exe}\ e'} \sigma'''$ 并且容易得出 $\sigma''' \xrightarrow{E\text{-}WAR} \sigma''$, 所以我们可以直接用归纳假设证明这些情况。
  - $P'' \neq P'$. 若 $P = P'$, 则该情况是平凡的。
    * 假设 $P' = \mathbb{P}[tid.r := E; C]$ 并且 $P = \mathbb{P}[r := x; r := E; C]$.
      · 如果 $P'' = \mathbb{P}'[tid.r := E; C]$, 则我们可以令 $P''' = \mathbb{P}'[r := x; r := E; C]$, 并且很容易找到相应的 $\sigma''' \xrightarrow{E\text{-}WAR} \sigma''$. 然后我们有 $P''' \xrightarrow{E\text{-}WAR} P''$, 所以我们就可以用归纳假设进行总结。
      · 若 $P'' = \mathbb{P}[tid.C]$，令 $P''' = \mathbb{P}[tid.C]$(意味着 (P, σ) 需要两步才能到达 (P''', σ''')). 我们可以像之前做的哪样进行总结。
    * 假设 $P' = \mathbb{P}[tid.While(r)C_1'; C_2]$ 并且 $P = \mathbb{P}[tid.While(r)C_1; C_2]$ where $C_1 \xrightarrow{E\text{-}WAR} C_2$. 若 $P'' = \mathbb{P}'[tid.While(r)C_1'; C_2]$，则我们按之前的分析可以立即得出结论。若 $P'' = \mathbb{P}[tid.C_1'; While(r)C_1'; C_2]$，则我们令 $P''' = \mathbb{P}[tid.C_1; While(r)C_1; C_2]$, 并且 $\sigma \xrightarrow{E\text{-}WAR} \sigma'$, 我们知道 $\sigma(tid)(r) = \sigma'(tid)(r)$, 并且在 σ 的缓存中并没有对寄存器 r 的写操作，所以我们可以有 $(P, \sigma) \to (P''', \sigma)$ 以及 $(P', \sigma') \to (P'', \sigma')$. 最后我们可以得出结论。

□

推论 B.1.1. *对于任意的 P, σ 以及 P', 若 $P \xrightarrow{E\text{-}WAR} P'$, $(P', \sigma) \longrightarrow^* (\textbf{skip}, \sigma')$ 和 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$ 并且 $\sigma' \overset{obser}{=\!=} \sigma''$。*

证明. 运用引理B.1.1, 我们有 $(P, \sigma) \sim (P', \sigma)$. 根据模拟关系的定义我们知道存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$, $\sigma' \overset{E\text{-}WAR}{=\!=\!=} \sigma''$ 和 $buff(\sigma') = buff(\sigma'') = \emptyset$. 所以我们根据 $\overset{obsv}{=\!=}$ 和 $\overset{E\text{-}WAR}{=\!=\!=}$ 的定义有，$\sigma' \overset{obsv}{=\!=} \sigma''$ 。 □

B.2　E-WBW 和 E-IR 正确性

在 E-WBW 和 E-IR 上定义出的模拟关系和 E-WAR 除了在 *Opti* 上的定义不同以外 (因为是不同的程序变换) 其他都一样。所以我们在这里只列出推论。

推论 B.2.1. *若对于任意的 P, σ 和 P', $P \xrightarrow{E\text{-}WBW} P'$, $(P', \sigma) \longrightarrow^* (\textbf{skip}, \sigma')$, 并且 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$ 和 $\sigma' \overset{obser}{=\!=} \sigma''$.*

证明. 证明过程和引理B.1.1一样。 □

推论 B.2.2. *对于任意的 P, σ 和 P', 若 $P \xrightarrow{E\text{-}IR} P'$, $(P', \sigma) \longrightarrow^* (\textbf{skip}, \sigma')$, 并且哦 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$ 并且 $\sigma' \overset{obser}{=\!=} \sigma''$.*

证明. 证明过程和引理B.1.1一样。 □

B.3　E-RAR 和 E-RAW 正确性

跟我们之前做的一样，我们通过模拟关系来证明 E-RAR. 我们只需要在等价性上做一些小改动即可

$$
\begin{array}{lcl}
elim(e_1, e_2) & \overset{\text{def}}{=} & e_1.ts = e_2.ts \\
 & & \wedge (e_1.\iota = r := [loc]_n \wedge e_2.\iota = r := n) \\
\sigma \overset{b_2}{=\!=} \sigma' & \overset{\text{def}}{=} & \text{sizeof}(buff(\sigma)) = \text{sizeof}(buff(\sigma)) \\
 & & \wedge \forall e \in buff(\sigma').\ (\exists e' \in buff(\sigma).e = e') \\
 & & \quad \vee (\exists e' \in buff(\sigma).elim(e', e)) \\
\sigma \overset{E\text{-}RAR}{=\!=\!=} \sigma' & \overset{\text{def}}{=} & \sigma \overset{r}{=} \sigma' \wedge \sigma \overset{b_2}{=\!=} \sigma' \wedge \sigma \overset{m}{=\!=} \sigma' \wedge \sigma \overset{l}{=} \sigma'
\end{array}
$$

定义 B.3.1 (The simulation relation)**.**

$$
\begin{array}{lcl}
(P, \sigma) \sim^n (P', \sigma') & \overset{def}{=} & (P \xrightarrow{E\text{-}RAR} P' \wedge \sigma \overset{E\text{-}RAR}{=\!=\!=} \sigma') \\
 & & \wedge \forall P'', \sigma''.\ (P', \sigma') \rightarrow (P'', \sigma'') \Rightarrow \\
 & & \quad \exists P''', \sigma'''.\ (P, \sigma) \rightarrow^* (P''', \sigma''') \\
 & & \quad \wedge (P''', \sigma''') \sim^{n-1} (P'', \sigma'') \\
(\textbf{skip}, \sigma) \sim^n (\textbf{skip}, \sigma') & \overset{def}{=} & \sigma \overset{E\text{-}RAR}{=\!=\!=} \sigma' \\
(P, \sigma) \sim^0 (P', \sigma') & \overset{def}{=} & P \xrightarrow{E\text{-}RAR} P' \wedge \sigma \overset{E\text{-}RAR}{=\!=\!=} \sigma' \\
(P, \sigma) \sim (P', \sigma') & \overset{def}{=} & \forall n.\ (P, \sigma) \sim^n (P', \sigma')
\end{array}
$$

引理 B.3.1. *对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E-RAR} P'$ 和 $\sigma \xrightarrow{E-RAR} \sigma'$, 并且我们假设 P' 的执行可以终止则我们有 $(P, \sigma) \sim (P', \sigma')$.*

证明. 我们需要证明 $\forall n.\ (P, \sigma) \sim^n (P', \sigma')$.
因此在 n 上做归纳假设:

- Base: $n = 0$. 基本情况可以直接根据定义证明。
- Inductive: 根据归纳假设，我们有:
  对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E\text{-}RAR} P'$ 并且 $\sigma \xrightarrow{E\text{-}RAR} \sigma'$ 并且 P' 的执行可以终止, 则 $(P, \sigma) \sim^n (P', \sigma')$ 成立。
  我们需要证明：
  对于任意的 P, σ, P', σ', 若 $P \xrightarrow{E\text{-}RAR} P'$ 并且 $\sigma \xrightarrow{E\text{-}RAR} \sigma'$ 以及 P' 的执行可以终止, 则 $(P, \sigma) \sim^{n+1} (P', \sigma')$ 成立
  证明如下:
  对于 $\forall P'', \sigma''.\ (P', \sigma') \to (P'', \sigma'')$, 我们需要对 P'' and P' 的结构进行讨论。
  - $P'' = P'$. 意味这一步中并没有发生从指令到缓存的变换，因此根据操作语义我们有 $(P', \sigma') \xrightarrow{(\textbf{BufA}\ ba, ts')} (P', \sigma'')$, 接下来我们根据 ba 进行讨论。
    * $ba = \textbf{Exe}\ e$. 在这种情况下，我们根据 e 来进行讨论，并且假设在 σ 中匹配的事件是 is e':
      · 若 $elim(e, e')$. 假设 $e.\iota = r_2 := r_1$ 并且 $e'.\iota = r_2 := x$. 由于 e 可以被执行，因此在缓存中不存在任何对 r_2 和 r_1 的写操作或者是对 r_2 的读操作（并且这些读操作的时间戳小于 e）所以在 σ 缓存中的事件 e' 可以被执行。并且,r_2 所读到的值在 σ 中仍然是可见的。因此我们可以证明此情形。
      · if $e = e'$. 这种情况是平凡的。
    * $ba = \textbf{Rep}\ e$. 这和 $ba = \textbf{Exe}\ e$ 是一样的。
  - $P'' \neq P'$. 这种情况的证明与引理B.1.1一样。

□

推论 B.3.1. *对于任意的 P, σ 并且 P', 若 $P \xrightarrow{E\text{-}RAR} P'$, $(P', \sigma) \longrightarrow^* (\textbf{skip}, \sigma')$, 并且 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$ 并且 $\sigma' \xrightarrow{obser} \sigma''$。*

证明. 如同我们在推论B.1.1中做的那样, 我们运用引理B.3.1来证明这一推论。
□

推论 B.3.2. *对于任意的 P, σ 和 P', 若 $P \xrightarrow{E\text{-}RAW} P'$, $(P', \sigma) \longrightarrow^* (\textbf{skip}, \sigma')$, 并且 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\textbf{skip}, \sigma'')$ 并且 $\sigma' \xrightarrow{obser} \sigma''$.*

证明. 同上。 □

B.4　调序变换的正确性

在下面这一小节中，我们将证明调序变换的正确性。注意到我们这里只证明了对没有冲突的**普通**变量的读操作进行调序。使用相同的方法，我们可以很简单的推论到其他调序情况的证明。当指令被包装成事件放入缓存以后，为了能够追踪事件的行为来构建观测等价性，我们需要往事件和指令上额外添加一些属性，这样可以用来帮助我们区分。我们假设抽象机知道哪两条语句将会被调序 (可以在程序中对要调序的指令添加标记来实现)。当要调序的指令在抽象机执行过程中被包装成事件的时候，事件将会额外多出一位标记为。并且，我们建立了一张表来存储调序指令被包装成事件的时间信息。

$$
\begin{array}{rcl}
\mathit{rec} & := & \langle \mathit{uid}, \mathsf{t}_1, \mathsf{t}_2 \rangle \\
\mathit{tab} & := & \mathit{rec} :: \mathit{tab}
\end{array}
$$

辅助定义：

$$
\mathit{offset}(\mathsf{t}, \mathit{tab}) := \begin{cases}
1 & \exists \mathit{rec} \in \mathit{tab}.\, \mathit{rec}.\mathsf{t}_1 \leqslant \mathsf{t} < \mathit{rec}.\mathsf{t}_2 \\
0 & (\exists \mathit{rec}_1, \mathit{rec}_2 \in \mathit{tab}.\mathit{rec}_2. \\
 & \qquad \mathit{uid} = \mathit{rec}_1.\mathit{uid} + 1 \\
 & \qquad \wedge \mathit{rec}_1.\mathsf{t}_2 < \mathsf{t} < \mathit{rec}_2.\mathsf{t}_1) \\
 & \vee \mathsf{t} < \mathit{minrec}(\mathit{tab}).\mathsf{t}_1 \\
 & \vee \mathsf{t} > \mathit{maxrec}(\mathit{tab}).\mathsf{t}_2 \\
-(\mathsf{t} - \mathit{rec}.\mathsf{t}_1) & \exists \mathit{rec} \in \mathit{tab}.\, \mathit{rec}.\mathsf{t}_2 = \mathsf{t}
\end{cases}
$$

$$
\begin{array}{rl}
\sigma \overset{\mathsf{b}_3}{=\!=} \sigma' \overset{\text{def}}{=} & \mathrm{sizeof}(\mathit{buff}(\sigma')) = \mathrm{sizeof}(\mathsf{B}(\sigma'))) \\
 & \wedge \forall e' \in \mathit{buff}(\sigma').\, \exists e \in \mathit{buff}(\sigma). \\
 & \quad e.\iota = e'.\iota \wedge e.\mathit{ts}.\mathit{tid} = e'.\mathit{ts}.\mathit{tid} \\
 & \quad \wedge e.\mathit{ts}.\mathsf{t} = e'.\mathit{ts}.\mathsf{t} + \mathit{offset}(e'.\mathit{ts}.\mathsf{t}, \sigma'.\mathit{tab})
\end{array}
$$

$$
\begin{aligned}
\sigma \overset{m_2}{=\!=} \sigma' \overset{\text{def}}{=} & \forall \ell \in \mathsf{dom}(\sigma').\ \sigma'.m(l) = n \Rightarrow \sigma.m(l) = n \\
& \wedge \sigma'.m(x) = h' \Rightarrow (\mathit{sizeof}(\sigma.m(x)) = \mathit{sizeof}(h') \\
& \quad \wedge \forall wv' \in h.\ \exists wv \in \sigma.m(x). \\
& \qquad wv.ts.t = wv'.ts.t + \mathit{offset}(wv'.ts.t, \sigma'.tab) \\
& \qquad \wedge wv.v = wv'.v \wedge wv.ts.tid = wv'.ts.tid \\
& \qquad \wedge wv.\mu = wv'.\mu) \\
& \wedge \forall syn' \in h.\ \exists syn \in \sigma.m(x). \\
& \quad syn.ts.t = syn'.ts.t + \mathit{offset}(syn'.ts.t, \sigma'.tab) \\
& \quad \wedge syn. = syn'. \\
& \wedge \mathsf{dom}(\sigma.m) = \mathsf{dom}(\sigma'.m)
\end{aligned}
$$

$$
\sigma \overset{\textit{R-RR}_1}{=\!=\!=} \sigma' \overset{\text{def}}{=} \sigma \overset{r}{=} \sigma' \wedge \sigma \overset{b_3}{=\!=} \sigma' \wedge \sigma \overset{m_2}{=\!=} \sigma' \wedge \sigma \overset{l}{=} \sigma'
$$

$$
\begin{aligned}
(P, \sigma) \overset{\textit{R-RR}_2}{=\!=\!=} (P', \sigma') \overset{\text{def}}{=} & \\
& P = \mathbb{P}[\mathit{tid}.C] \wedge P' = \mathbb{P}[\mathit{tid}.r := x; C] \\
& \mathit{sizeof}(\mathit{buff}(\sigma)) = \mathit{sizeof}(\mathit{buff}(\mathtt{stateRR})) + 1 \\
& \wedge \forall e' \in \mathit{buff}(\sigma').\ \exists e \in \mathit{buff}(\sigma). \\
& \qquad e.\iota = e'.\iota \wedge e.ts.tid = e'.ts.tid \\
& \qquad \wedge e.ts.t = e'.ts.t + \mathit{offset}(e'.ts.t, \sigma'.tab) \\
& \wedge \exists e \in \mathit{buff}(\sigma).\ e.\iota = r := x; \\
& \qquad \wedge \neg \exists e' \in \mathit{buff}(\sigma').\ e.ts.tid = e'.ts.tid \\
& \qquad\quad \wedge e.ts.t = e'.ts.t + \mathit{offset}(e'.ts.t, \sigma'.tab) \\
& \wedge \mathit{maxrec}(\sigma'.tab).t_2 = \infty \\
& \wedge \sigma \overset{r}{=} \sigma' \wedge \sigma \overset{l}{=} \sigma' \wedge \sigma \overset{m}{=\!=} \sigma' \wedge \sigma.t = \sigma'.t + 1
\end{aligned}
$$

定义 B.4.1 (The simulation relation)**.**

$$
\begin{aligned}
(P, \sigma) \sim^n (P', \sigma') \overset{\mathit{def}}{=} & ((P \xrightarrow{r} P' \wedge \sigma \overset{\textit{R-RR}_1}{=\!=\!=} \sigma') \\
& \qquad \vee (P, \sigma) \overset{\textit{R-RR}_2}{=\!=\!=} (P', \sigma')) \\
& \quad \wedge \forall P'', \sigma''.\ (P', \sigma') \to (P'', \sigma'') \Rightarrow \\
& \qquad \exists P''', \sigma'''.\ (P, \sigma) \to^* (P''', \sigma''') \\
& \qquad\quad \wedge (P''', \sigma''') \sim^{n-1} (P'', \sigma''_r) \\
(\mathbf{skip}, \sigma) \sim^n (\mathbf{skip}, \sigma') \overset{\mathit{def}}{=} & \sigma \overset{\textit{R-RR}_1}{=\!=\!=} \sigma' \\
(P, \sigma) \sim^0 (P', \sigma') \overset{\mathit{def}}{=} & (P \xrightarrow{r} P' \wedge \sigma \overset{\textit{R-RR}_1}{=\!=\!=} \sigma') \\
& \qquad \vee (P, \sigma) \overset{\textit{R-RR}_2}{=\!=\!=} (P', \sigma') \\
(P, \sigma) \sim (P', \sigma') \overset{\mathit{def}}{=} & \forall n.\ (P, \sigma) \sim^n (P', \sigma')
\end{aligned}
$$

引理 B.4.1. 对于任意的 P, σ, P', σ', 若 $P \xrightarrow{\textit{R-RR}} P'$ 并且 $\sigma \overset{\textit{R-RR}_1}{=\!=\!=} \sigma'$ 或者 $(P, \sigma) \overset{\textit{R-RR}_2}{=\!=\!=} (P, \sigma')$，同时我们假设 P' 的执行可以终止，那么我们有 $(P, \sigma) \sim (P', \sigma')$。

证明.　• Base: 我们可以根据 $\sim^0$ 的定义直接有 $(P,\sigma) \sim^0 (P',\sigma')$。

• Inductive: 根据归纳假设，我们有
对于任意的 P,σ,P',σ', 若 $P \xrightarrow{R\text{-}RR} P'$ 并且 $\sigma \xrightarrow{R\text{-}RR_1} \sigma'$ 或者 $(P,\sigma) \xrightarrow{R\text{-}RR_2} (P,\sigma')$，并且 P' 的执行可以终止，那么我们有 $(P,\sigma) \sim^n (P',\sigma')$。我们需要证明：
对于任意的 P,σ,P',σ', 若 $P \xrightarrow{R\text{-}RR} P'$ 并且 $\sigma \xrightarrow{R\text{-}RR_1} \sigma'$ 或者 $(P,\sigma) \xrightarrow{R\text{-}RR_2} (P,\sigma')$，并且 P' 的执行可以终止，那么我们有 $(P,\sigma) \sim^{n+1} (P',\sigma')$。证明如下所示：

– $P \xrightarrow{R\text{-}RR} P' \wedge \sigma \xrightarrow{R\text{-}RR_1} \sigma'$
对于任意的 P'',σ'' 使得 $(P',\sigma') \to (P'',\sigma'')$, 我们可以对 P'' 和 P' 的结构进行讨论:

* $P'' = P'$. 意味这一步中并没有发生从指令到缓存的变换，因此根据操作语义我们有 $(P',\sigma') \xrightarrow{(\textbf{BufA}\ ba,ts')} (P',\sigma'')$, 接下来我们根据 ba 进行讨论。

· $ba = \textbf{Exe}\ e$. 在这种情况下，我们根据 e 来进行讨。

1. e 是普通变量的写事件。假设 $e = \langle ts, x := k\rangle$。在 σ 中我们可以找到 e_1 使得 $e_1 \approx e$。并且就像我们之前分析的那样我们知道在 σ 中 e_1 是可以被执行的。假设 $\sigma \xrightarrow{\textbf{Exe}\ e_1} \sigma'''$, 我们可以有 $\sigma''' \xrightarrow{m} stateRR''$. 所以 $\sigma''' \xrightarrow{R\text{-}RR_1} stateRR''$。因此我们可以得出结论。
2. e 是普通变量的读事件。假设 $e = \langle ts, r := x\rangle$. 并且我们可以在 σ 的缓存中找到 e_1 使得 $e_1 \approx e$。由于 $\sigma \xrightarrow{m} \sigma'$, 我们可以知道在 σ 中 e 读到的值一定对于 e_1 是可见的。假设 $\sigma \xrightarrow{\textbf{Exe}\ e_1} \sigma'''$, 则我们有 $\sigma''' \xrightarrow{r} \sigma''$. 因此我们可以得出结论。
3. e 是一个同步变量的写事件。这个情况和普通变量的写操作一样。
4. e 是一个同步变量的读事件。这个情况和普通变量的读操作一样。
5. e 是一个释放资源事件. 假设 $e = \langle ts, \textbf{unlock}\, l\rangle$, 我们可以在 σ 中找到 e_1 使得 $e_1 \approx e$。由于 $\sigma \xrightarrow{l} \sigma'$，并且假设 $\sigma \xrightarrow{\textbf{Exe}\ e_1} \sigma'''$, 我们有 $\sigma''' \xrightarrow{l} \sigma''$. 因此我们可以得出 $\sigma''' \xrightarrow{R\text{-}RR_1} \sigma''$. 然后我们就能证明此情况。

· $ba = \textbf{Rep}\ e$. 和 $ba = \textbf{Exe}\ e$ 的情况一样。

* $P'' \neq P'$. 如果 $P = P'$, 则是平凡的，因此我们讨论剩余的情况：

· 假设 $P' = \mathbb{P}[tid.\mathbb{E}[r_1 := x_1; r_2 := x_2]]$ 和 $P = \mathbb{P}[tid.\mathbb{E}[r_2 := x_2; r_1 := x_1]]$. 对于任意的 P'',σ'', 若 $(P',\sigma') \to (P'',\sigma'')$, 我们根据 P'' 的结构进行讨论：

1. $P'' = \mathbb{P}'[tid.\mathbb{E}[r_1 := x_1; r_2 := x_2]]$ 或 $P'' = \mathbb{P}[tid.\mathbb{E}'[r_1 := x_1; r_2 := x_2]]$. 这种情况也是平凡的，因为我们只需要令 $(P,\sigma) \to (P''',\sigma''')$ 并且 $P''' = \mathbb{P}'[tid.\mathbb{E}[r_2 := x_2; r_1 := x_1]]$ 或者 $P''' = \mathbb{P}[tid.\mathbb{E}'[r_2 := x_2; r_1 := x_1]]$ 即可.

2. $P'' = \mathbb{P}[tid.\mathbb{E}[r_2 := x_2]]$. 在这种情况下，我们令:
$(P, \sigma) \rightarrow (\mathtt{context}[tid.\mathbb{E}[r_2 := x_2;], \sigma^0) \rightarrow (P''', \sigma''')$ 并且 $P''' = \mathbb{P}[tid.\mathbb{E}[]]$.

– $(P, \sigma) \xrightarrow{R\text{-}RR_2} (P', \sigma')$
对于任意的 P'', σ'' 使得 $(P', \sigma') \rightarrow (P'', \sigma'')$, 我们可以根据 P'' 和 P' 进行讨论。

* $P'' = P'$. 意味这一步中并没有发生从指令到缓存的变换，因此根据操作语义我们有 $(P', \sigma') \xrightarrow{(\mathbf{BufA}\ ba, ts')} (P', \sigma'')$, 剩下的证明就跟之前我们证过的一样。
* $P'' \neq P'$. 若 $P = P'$, 这种情况是平凡的，只要令 $P' = \mathbb{P}[tid.\mathbb{E}[r := x]]$ 和 $P = \mathbb{P}[tid.\mathbb{E}[]]$.
  1. 若 $P'' = \mathbb{P}[[tid.\mathbb{E}[]]$, 则我们让 $P''' = P$ 并且诶 $\sigma''' = \sigma$, 那么我们可以知道 $P''' \xrightarrow{r} P'' \wedge \sigma''' \xrightarrow{R\text{-}RR_1} \sigma''$.
  2. $P'' = \mathbb{P}'[[tid.\mathbb{E}[r := x]]$. 令 $P''' = \mathbb{P}'[tid.\mathbb{E}[]]$, 则有 $(P, \sigma) \rightarrow (P''', \sigma''')$ 并且
  $(P''', \sigma''') \xrightarrow{R\text{-}RR_2} (P'', \sigma'')$.

□

推论 B.4.1. *对于任意的 P, σ 以及 P', 若 $P \xrightarrow{R\text{-}RR} P'$, $(P', \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$ 和 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma'')$ 并且 $\sigma' \xrightarrow{obser} \sigma''$。*

证明. 如同我们在推论B.1.1中做的那样, 我们运用引理B.4.1来证明这一推论。
□

用同样的方法，我们可以证出

推论 B.4.2. *对于任意的 P, σ 以及 P', 若 $P \xrightarrow{s} P'$, $(P', \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$ 和 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma'')$ 并且 $\sigma' \xrightarrow{obser} \sigma''$。*

证明. 同上。 □

B.5 I-IR 的正确性

如我们之前在第三章中所说的那样，冗余的读操作引入可以被定义为冗余读操作删除的（E-IR）的反操作。

$$\frac{}{r := \mathsf{E}; \xrightarrow{I\text{-}RR} r := x; r := \mathsf{i}} \text{(I-IR)}$$

我们还是使用表格来记录当冗余读操作被封装成事件时的逻辑时间。

$$\begin{aligned} rec &:= \langle uid, \mathsf{t}_1, \mathsf{t}_2 \rangle \\ tab &:= rec :: tab \end{aligned}$$

举个例子，比如在表格中当前最大的索引值是 uid, 并且我们把目标程序 $\mathsf{r} := [\mathsf{l}]_n; \mathsf{r} := \mathsf{n}$ 转换成两个不同的事件，事件的时间分别是 t_1 和 t_2, 则表格中记录添加一条记录，为 $\langle uid + 1, \mathsf{t}_1, \mathsf{t}_2 \rangle$

辅助定义

$$offset(\mathtt{t}, tab) := \begin{cases} -\mathtt{n} & \exists rec \in tab.\, rec.\mathtt{t}_1 \leqslant \mathtt{t} <= rec.\mathtt{t}_2 \wedge \mathtt{n} = rec.uid \\ & \vee(\mathtt{t} > maxrec(tab).\mathtt{t}_1 \wedge maxrec(tab).uid = \mathtt{n}) \\ 0 & \mathtt{t} < minrec(tab).\mathtt{t}_1 \end{cases}$$

$$\begin{aligned} \sigma \overset{\mathtt{b}_4}{=\!=} \sigma' \quad \overset{\text{def}}{=} \quad & \forall \mathtt{e}' \in buff(\sigma').\, \exists (rec \in \sigma'.tab.\, \mathtt{e}'.\mathtt{t} = rec.\mathtt{t}_1 \\ & \qquad \wedge \mathtt{e}'.\iota = \mathtt{r} = [\mathtt{loc}]_\mathtt{n}) \\ & \vee (\exists \mathtt{e} \in buff(\sigma).\, \mathtt{e}.\iota = \mathtt{e}'.\iota \wedge \mathtt{e}.ts.tid = \mathtt{e}.'ts.tid = \mathtt{t} \\ & \qquad \wedge \mathtt{e}.ts.\mathtt{t} = \mathtt{e}'.ts.\mathtt{t} + offset(\mathtt{t}, \sigma.tab)) \end{aligned}$$

$$\begin{aligned} \sigma \overset{\mathtt{r}'}{=\!=} \sigma' \quad \overset{\text{def}}{=} \quad & ((\exists rec \in \sigma'.tab.\, \exists! \mathtt{e}' \in buff(\sigma').\, \mathtt{e}'.\mathtt{t} = rec.\mathtt{t}_2 \\ & \qquad \wedge \mathtt{e}'.ts.tid = \mathtt{i}' \wedge \mathtt{e}'.\iota = \mathtt{r}' := \mathtt{x} \\ & \qquad \wedge \neg \exists \mathtt{e}'' \in buff(\sigma).\, \mathtt{e}''.\mathtt{t} = rec.\mathtt{t}_1) \\ & \Rightarrow (\forall \mathtt{i} \neq \mathtt{i}'.\, \sigma.tq(\mathtt{i}).rf = \sigma'.tq(\mathtt{i}).rf \\ & \qquad \wedge \forall \mathtt{r}! = \mathtt{r}' \sigma.tq(\mathtt{i}).rf(\mathtt{r}) = \sigma'.tq(\mathtt{i}).rf(\mathtt{r}))) \\ & \vee \sigma \overset{\mathtt{r}}{=} \sigma' \end{aligned}$$

$$\sigma \overset{I\text{-}RR}{=\!=} \sigma' \overset{\text{def}}{=} \sigma \overset{\mathtt{r}'}{=\!=} \sigma' \wedge \sigma \overset{\mathtt{b}_4}{=\!=} \sigma' \wedge \sigma \overset{\mathtt{m}}{=\!=} \sigma' \wedge \sigma \overset{\mathtt{l}}{=} \sigma'$$

定义 B.5.1 (模拟关系)**.**

$$\begin{aligned} (\mathtt{P}, \sigma) \sim^{\mathtt{n}} (\mathtt{P}', \sigma') \quad & \overset{def}{=} \quad (\mathtt{P} \xrightarrow{I\text{-}RR} \mathtt{P}' \wedge \sigma \overset{I\text{-}RR}{=\!=} \sigma') \\ & \qquad \wedge \forall \mathtt{P}'', \sigma''.\, (\mathtt{P}', \sigma') \to (\mathtt{P}'', \sigma'') \Rightarrow \\ & \qquad\quad \exists \mathtt{P}''', \sigma'''.\, (\mathtt{P}, \sigma) \to^* (\mathtt{P}''', \sigma''') \\ & \qquad\qquad \wedge (\mathtt{P}''', \sigma''') \sim^{\mathtt{n}-1} (\mathtt{P}'', \sigma'') \\ (\mathbf{skip}, \sigma) \sim^{\mathtt{n}} (\mathbf{skip}, \sigma') \quad & \overset{def}{=} \quad \sigma \overset{I\text{-}RR}{=\!=} \sigma' \\ (\mathtt{P}, \sigma) \sim^{0} (\mathtt{P}', \sigma') \quad & \overset{def}{=} \quad \mathtt{P} \xrightarrow{I\text{-}RR} \mathtt{P}' \wedge \sigma \overset{I\text{-}RR}{=\!=} \sigma' \\ (\mathtt{P}, \sigma) \sim (\mathtt{P}', \sigma') \quad & \overset{def}{=} \quad \forall \mathtt{n}.\, (\mathtt{P}, \sigma) \sim^{\mathtt{n}} (\mathtt{P}', \sigma') \end{aligned}$$

引理 B.5.1. *对于任意的 $\mathtt{P}, \sigma, \mathtt{P}', \sigma'$, 若 $\mathtt{P} \xrightarrow{I\text{-}RR} \mathtt{P}'$ 和 $\sigma \overset{I\text{-}RR}{=\!=} \sigma'$, 并且我们假设 $\mathtt{P}'$ 的执行可以终止则我们有 $(\mathtt{P}, \sigma) \sim (\mathtt{P}', \sigma')$.*

证明.　• Base: $(\mathtt{P}, \sigma) \sim^{0} (\mathtt{P}', \sigma')$ 平凡可根据定义得到。

- Inductive: 根据归纳假设，我们有:
  对于任意的 $\mathtt{P}, \sigma, \mathtt{P}', \sigma'$, 若 $\mathtt{P} \xrightarrow{I\text{-}RR} \mathtt{P}'$ 并且 $\sigma \overset{I\text{-}RR}{=\!=} \sigma'$ 并且 $\mathtt{P}'$ 的执行可以终止, 则 $(\mathtt{P}, \sigma) \sim^{\mathtt{n}} (\mathtt{P}', \sigma')$ 成立。
  我们需要证明：
  对于任意的 $\mathtt{P}, \sigma, \mathtt{P}', \sigma'$, 若 $\mathtt{P} \xrightarrow{I\text{-}RR} \mathtt{P}'$ 并且 $\sigma \overset{I\text{-}RR}{=\!=} \sigma'$ 以及 $\mathtt{P}'$ 的执行可以终止, 则 $(\mathtt{P}, \sigma) \sim^{\mathtt{n}+1} (\mathtt{P}', \sigma')$ 成立
  证明如下:
  对于 $\forall \mathtt{P}'', \sigma''.\, (\mathtt{P}', \sigma') \to (\mathtt{P}'', \sigma'')$, 我们需要对 $\mathtt{P}''$ and $\mathtt{P}'$ 的结构进行讨论。
  - $\mathtt{P}'' = \mathtt{P}'$. 意味这一步中并没有发生从指令到缓存的变换，因此根据操作语义我们有 $(\mathtt{P}', \sigma') \xrightarrow{(\mathbf{BufA}\ ba, ts')} (\mathtt{P}', \sigma'')$, 接下来我们根据 ba 进行讨论。
    * $ba = \mathbf{Exe}\ \mathtt{e}$. 在这种情况下，我们根据 $\mathtt{e}$ 来进行讨论，并且假设在 σ 中匹配的事件是 is $\mathtt{e}'$:

1. e 是一个普通变量的写事件. 假设 $e = \langle ts, x := k\rangle$. 我们可以在 σ 中找到 e_1 使得 $e_1 \approx e$。就如我们之前分析的那样，我们知道在 σ 中的 e_1 是可以被执行的，即当前没有任何它所依赖的事件存在于缓存中。假设 $\sigma \xrightarrow{\mathbf{Exe}\ e_1} \sigma'''$. 我们有 $\sigma''' \overset{m}{=\!=} stateRR''$。因此我们知道 $\sigma''' \overset{R\text{-}RR_1}{=\!=\!=} stateRR''$，由此我们容易得出结论
2. e 是一个普通变量的读事件。
   (a) $\exists rec \in \sigma'.tab.\ e.ts.t = rec.t_1$. 这种情况以为着 e 是冗余的读操作并且我们让 $P''' = P \wedge \sigma''' = \sigma$, 所以根据 $\overset{r_2}{=\!=}$ 的定义, 我们有 $\sigma''' \overset{I\text{-}RR}{=\!=\!=} \sigma''$。由此我们可以证明该情况
   (b) $\neg\exists rec \in \sigma'.tab.\ e.ts.t = rec.t_1$. 假设 $e = \langle ts, r := x\rangle$, 以及 e_1 在 σ 中，使得 $e_1 \approx e$。由于 $\sigma \overset{m}{=\!=} \sigma'$, 我们有 e 读到的值对 σ 中的 e_1 是可见的。假设 $\sigma \xrightarrow{\mathbf{Exe}\ e_1} \sigma'''$。所以我们有 $\sigma''' \overset{r_2}{=\!=} \sigma''$。由此我们可以证明该情况。
3. e 是一个同步变量的写事件。这种情况的证明过程和普通变量写事件相似。
4. e 是一个同步变量的读事件。这种情况的证明过程和普通变量读事件相似。
5. e 是一个释放锁资源事件。假设 $e = \langle ts, \mathbf{unlock} l\rangle$. 并且我们在 σ 中可以找到 e_1 使得 $e_1 \approx e$。由于 $\sigma \overset{l}{=\!=} \sigma'$ 并且假设 $\sigma \xrightarrow{\mathbf{Exe}\ e_1} \sigma'''$, 我们有 $\sigma''' \overset{l}{=\!=} \sigma''$. 因此 $\sigma''' \overset{I\text{-}RR_1}{=\!=\!=} \sigma''$。所以我们可以得出结论。

* $ba = \mathbf{Rep}\ e$. 和 $ba = \mathbf{Exe}\ e$ 类似。

– $P'' \neq P'$. 若 $P = P'$, 这种情况是平凡的。

* 假设 $P' = \mathbb{P}[tid.\mathbb{E}[r := x; r := n;]]$ 并且 $P = \mathbb{P}[tid.\mathbb{E}[r := n]]$. 对于任意的 P'', σ'', 若 $(P', \sigma') \to (P'', \sigma'')$, 我们可以根据 P'' 的结构进行讨论:
   1. $P'' = \mathbb{P}'[tid.\mathbb{E}[r := x; r := n;]]$ 或者 $P'' = \mathbb{P}[tid.\mathbb{E}'[r := x; r := n;]]$. 这种情形是平凡的, 由于我们只需要令 $(P, \sigma) \to (P''', \sigma''')$ 和 $P''' = \mathbb{P}'[tid.\mathbb{E}[r_2 := x_2; r_1 := x_1]]$ 或者 $P''' = \mathbb{P}[tid.\mathbb{E}'[r_2 := x_2; r_1 := x_1]]$.
   2. $P'' = \mathbb{P}[tid.\mathbb{E}[r]]$. 在这种情况下, 令 $P''' = P' \wedge \sigma''' = \sigma$ 可得结论

□

推论 B.5.1. *对于任意的 P, σ 以及 P', 若 $P \xrightarrow{I\text{-}RR} P'$, $(P', \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$ 和 $buff(\sigma') = \emptyset$, 则存在 σ'' 使得 $(P, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma'')$ 并且 $\sigma' \overset{obser}{=\!=\!=} \sigma''$。*

证明. 如同我们在推论B.1.1中做的那样, 我们运用引理B.4.1来证明这一推论。
□

附录 C　TSO 和 ATSO 精化关系证明

在这一附录中，我们将证明引理5.5.2。

证明. 假设 $P = tid_1.\mathrm{C}_1 \parallel \cdots \parallel tid_i.\mathrm{C}_i \parallel \ldots tid_n.\mathrm{C}_n$ 以及 $P' = tid_1.\mathrm{C}_1 \parallel \cdots \parallel tid_i.\mathrm{C}'_i \parallel \ldots tid_n.\mathrm{C}_n$. 我们有 $(P,\sigma) \rightsquigarrow (P',\sigma')$. 运用 “State Transition” 以及 “Program Step”(图 5.3), 我们有 $(\mathrm{C}_i,(m,rf,buf)) \rightsquigarrow (\mathrm{C}'_i,(m',rf',buf'))$，其中 $\sigma.m = m, \sigma'.m = m', \sigma.thds(tid_i) = (rf,\mathtt{buf})$ 并且
$\sigma'.thds(tid_i) = (rf',\mathtt{buf}')$. 我们可以根据 C_i 来分情况讨论. 我们需要讨论以下几种情况：

1. $\mathrm{C}_i = \mathrm{C}'_i$. 根据图 5.3中的操作语义, 我们知道，这一步状态转移要么是 “Write from write buffer to memory”，或者 “Double Write from write buffer to memory”。因为这两种情况的证明基本相同，因此我们在这边只证明前者。我们知道 $buf = \mathrm{o} :: buf'$。假设 $\Sigma.thds(tid_i) = (m_l, rf_1, buf_1)$。根据 $\sigma \approx \Sigma$ 的定义, 我们有 $buf_1 \subseteq buf$，其中 $buf_1 = \mathrm{o}' :: buf'_1$. 若 $\mathrm{o} \neq \mathrm{o}'$, 则我们令 $\Sigma = \Sigma'$ 并且能够立即推导出结论。若 $\mathrm{o} = \mathrm{o}'$, 则我们可以应用图 5.7中的 “Write from write buffer to memory” 的状态转换来得到 Σ'. 容易推知 $\blacktriangleright_I \Sigma = \blacktriangleright_I \Sigma'$ 以及 $\blacktriangleright \sigma = \blacktriangleright \sigma'$, 并且 $\Sigma'.T(tid_i).buf = buf'_1$。所以我们知道 $buf'_1 \subseteq buf'$. 由此我们能够得出结论 $\sigma' \approx \Sigma'$

2. $\mathrm{C}_i \neq \mathrm{C}'_i$. 我们需要分以下几种情况:

   (a) $\mathrm{C}_i = (r := [E]; \mathrm{C}'_i)$. 我们细化为两种情况。

   - 从内存中读取。我们有 $rf' = rf[r \mapsto v]$，其中 $m(\llbracket E \rrbracket_{rf}) = v, \llbracket E \rrbracket_{rf} \notin \mathsf{dom}(buf), m = m'$ 并且 $buf = buf'$. 由于 $\sigma \approx \Sigma$, 我们有 $rf = rf_1$ 以及 $buf_1 \subseteq buf$ where $\Sigma.T(tid_i) = (m_l, rf_1, buf_1)$. 因此我们知道 $\llbracket E \rrbracket_{rf_1} \notin \mathsf{dom}(buf_1)$. 由此我们可以运用图 5.7中的 “Read from memory” 来得到 Σ'。所以我们有 $rf'_1 = rf_1[r \mapsto v]$，其中 $\Sigma'.T(tid_i) = (m_l', rf'_1, buf'_1)$。我们还知道 $buf'_1 = buf_1$ 以及 $buf = buf'$。因此我们有 $buf'_1 \subseteq buf'$. 根据操作语义，我们知道 $(\blacktriangleright \sigma).m = (\blacktriangleright \sigma').m$ 并且 $(\blacktriangleright_I \Sigma.m \uplus \mathsf{local}(\blacktriangleright \Sigma)) = (\blacktriangleright \Sigma'.m \uplus \mathsf{local}(\blacktriangleright \Sigma'))$. 根据 $\approx$ 的定义我们还有 $\blacktriangleright \sigma.m = (\blacktriangleright_I \Sigma.m \uplus \mathsf{local}(\blacktriangleright_I \Sigma)$, 所以我们可以得出结论 $\sigma' \approx \Sigma'$。
   - 从缓存中读取。我们有 $rf' = rf[r \mapsto v]$，其中 $buf(\llbracket E \rrbracket_{rf}) = v, m = m'$ 并且 $buf = buf'$。由于 $\sigma \approx \Sigma$, 我们有 $rf = rf_1$ 以及 $buf_1 \subseteq buf$，其中 $\Sigma.T(tid_i) = (m_l, rf_1, buf_1)$. 由于 $\llbracket E \rrbracket_{rf} \notin buf_1$, 我们可以应用图 5.7中 “Read from memory” 状态转换来得到 Σ'。由于 $\sigma \approx \Sigma$, 我们有 $buf(\llbracket E \rrbracket_{rf}) = m_l(\llbracket E \rrbracket_{rf})$. 所以我们知道 $rf'_1 = rf_1[r \mapsto v]$，其中 $\Sigma'.T(tid_i) = (m_l', rf'_1, buf'_1)$. 我们还知道 $buf'_1 = buf_1$ 并且 $buf = buf'$。由此可得 $buf'_1 \subseteq buf'$。根据操作语义，我们有 $(\blacktriangleright \sigma).m = (\blacktriangleright \sigma').m$ 并且 $(\blacktriangleright_I \Sigma.m \uplus \mathsf{local}(\blacktriangleright_I \Sigma)) = (\blacktriangleright \Sigma'.m \uplus \mathsf{local}(\blacktriangleright_I \Sigma'))$。根据 $\approx$ 的定义, 我们可以有 $\sigma' \approx \Sigma'$. 若 $\llbracket E \rrbracket_{rf} \in buf_1$, 我们则可以应用图 5.7中 “Read from write buffer” 状态转换来得到 Σ'。由于 $\sigma \approx \Sigma$, 我们有 $(\blacktriangleright \sigma).m(\llbracket E \rrbracket_{rf}) = (\blacktriangleright_I \Sigma).m(\llbracket E \rrbracket_{rf})$。如同我们之前分析的那样，我们可以证明该情形。

(b) $\mathrm{C_i} = (([E] := E')_\alpha); \mathrm{C_i'}$. 我们有 $m = m'$, $rf = rf'$ 并且 $buf' = (a, v)_\alpha :: buf$，其中 $[\![E]\!]_{rf} = a$ 以及 $[\![E']\!]_{rf} = v$. 若 $a \in (\blacktriangleright \Sigma).m_s$, 我们则可以应用图 5.7中的 “Write to write buffer” 的状态转换来得到 Σ'. 根据操作语义我们有 $buf_1' = (a, v) :: buf_1$ 其中 $\Sigma.T(tid_\mathrm{i}) = (m_l, rf_1, buf_1)$. 由于 $\sigma \approx \Sigma$, 我们有 $buf_1 \subseteq buf$ 以及 $rf = rf_1$. 因此我们知道 $buf_1' \subseteq buf'$ and $rf' = rf_1'$，其中 $\Sigma'.T(tid_\mathrm{i}) = (m_l', rf_1', buf_1')$。我们还有 $\blacktriangleright \sigma'.m = (\blacktriangleright_\mathrm{I} \Sigma'.m \uplus \mathsf{local}(\blacktriangleright_\mathrm{I} \Sigma))$。根据 $\approx$ 的定义, 我们可以有 $\sigma \approx \Sigma'$。若 $a \notin (\blacktriangleright_\mathrm{I} \Sigma).m_s$, 我们可以应用图 5.7中 “Direct write to local” 状态转换来得到 Σ'。由此我们有 $buf_1 = buf_1'$ 和 $rf_1 = rf_1'$，其中 $\Sigma.T(tid_\mathrm{i}) = (m_l, rf_1, buf_1)$ 并且 $\Sigma'.T(tid_\mathrm{i}) = (m_l', rf_1', buf_1')$. 因此我们知道 $buf_1' \subseteq buf \subseteq buf'$。所以我们能够证明该情形。

(c) $\mathrm{C_i} = (\langle [E] := E', [E_1] = E_1' \rangle); \mathrm{C_i'}$. 这个证明过程和 Case (b) 相同.

(d) $\mathrm{C_i} = r = \mathsf{CAS}(E_1, E_2, E_3)_\alpha; \mathrm{C_i'}$. 若 $m(a) \neq v$, 其中 $[\![E_1]\!]_{rf} = a$ 并且 $[\![E_2]\!]_{rf} = v$, 我们有 $(\mathrm{C_i}, (m, rf, buf)) \rightsquigarrow (\mathrm{C_i'}, (m', rf', buf'))$ 是图 5.3中 “CAS-FALSE” 状态转换。根据操作语义，我们有 $buf = \mathbf{nil}$。因为 $\sigma \approx \Sigma$, 我们有 $buf_1 = \mathbf{nil}$ 以及 $rf = rf_1$，其中 $\Sigma.T(tid_\mathrm{i}) = (m_l, rf_1, buf_1)$. 所以我们可以用图 5.7中的 “CAS-FALSE” 状态转换规则来得到 Σ'。同样，根据操作语义，我们有 $\Sigma.m = \Sigma'.m$, $m_l = m_l'$, $rf_1' = rf_1\,[\mathrm{r} \mapsto 0]$ 并且 $buf_1 = buf_1' = \mathbf{nil}$。所以我们知道 $rf_1' = rf'$ 并且 $buf_1' \subseteq buf'$. 我们同样有 $\blacktriangleright \sigma'.m = ((\blacktriangleright \Sigma').m \uplus \mathsf{local}(\blacktriangleright \Sigma))$。因此我们能够得出结论 $\sigma' \approx \Sigma'$. 若 $m(a) = v$，其中 $[\![E_1]\!]_{rf} = a$ 并且 $[\![E_2]\!]_{rf} = v$, 我们知道 $(\mathrm{C_i}, (m, rf, buf)) \rightsquigarrow (\mathrm{C_i'}, (m', rf', buf'))$ 是 图 5.3中的 “CAS-TRUE”。根据操作语义我们有 $buf = \mathbf{nil}$. 因为 $\sigma \approx \Sigma$, 我们有 $buf_1 = \mathbf{nil}$ 并且 $rf = rf_1$ ，其中 $\Sigma.T(tid_\mathrm{i}) = (m_l, rf_1, buf_1)$. 所以我们可以应用图 5.7中的 “CAS-TRUE with ownership transfer” 来得到 Σ'. 综上，我们可以得出结论。

□

附录 D　逻辑在 ATSO 上可靠性主引理的证明

在这一附录中，我们将证明引理5.5.6。

证明. 我们通过对 $R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\}$ 进行归纳假设来证明。

1.
$$\frac{}{R, G, \mathrm{I} \vdash \{tp\}\mathbf{skip}\{tp\}}\ (\text{Skip})$$

根据定义5.5.3, 我们需要证明，对于任意的 Δ, 若 $\Delta \models tp$ and $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$, 则 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{n}} (\mathbf{skip}, \Delta) : (\mathcal{G}, tp)$ 对于任意 $\mathfrak{n}$ 成立。

我们可以对 $\mathfrak{n}$ 进行归纳假设:

- Base case: $\mathfrak{n} = 0$。这种情况是平凡可证的。
- Suppose $\mathfrak{n} = \mathfrak{j} + 1$。展开定义5.5.4我们有四个子目标：
  (a) $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$;
  我们可以直接通过前提得到结论。
  (b) 若 $\mathrm{C} = \mathbf{skip}$, 则 $\Delta \models_{\mathrm{I}} tp$;
  这个子目标仍然是前提之一，直接可证。
  (c) $\mathrm{C} \neq \mathbf{skip}$, 则存在 C' 并且 Δ' 使得 $\mathrm{I} \vdash (\mathrm{C}, \Delta) \rightsquigarrow (\mathrm{C}', \Delta')$, $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{j}} (\mathrm{C}', \Delta') : (\mathcal{G}, tq)$;
  由于 $\mathrm{C} = \mathbf{skip}$, 直接可证。
  (d) 对于任意的 m' 使得 $(\Delta.m_s, m') \in \mathcal{R}$, 我们有 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{j}} (\mathbf{skip}, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tp)$;
  由于 $(\Delta.m_l, m') \in \mathcal{R}$ 并且 $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$, 我们可以应用推论5.5.4知道 $\mathcal{R} \models_{\mathrm{I}} (m', \Delta.m_l, \Delta.rf, \Delta.buf) : \mathcal{G}$。由于 $\mathsf{sta}(tp, R * \mathsf{Id})$ 并且 $(\Delta.m_l, m') \in \mathcal{R}$, 我们有 $(m', \Delta.m_l, \Delta.rf, \Delta.buf) \models_{\mathrm{I}} tp$。所以我们知道 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{j}} (\mathbf{skip}, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tp)$ 根据归纳假设可证。

2.
$$\frac{}{\mathbf{Emp}, \mathbf{Emp}, \mathbf{emp} \vdash \{E_1 \mapsto \_\}[E_1] = E_2\{E_1 \mapsto E_2\}}\ (\text{L-ASSN})$$

根据定义5.5.3, 我们需要证明对于任意的 Δ, 若 $\Delta \models_{\mathrm{I}} tp * (\mathsf{E}_1 \mapsto \_)$ 并且 $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$, 则 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{n}} (\mathrm{C}, \Delta) : (\mathcal{G}, tq)$ 对于任意 $\mathfrak{n}$ 成立, 其中 $\mathrm{C} = [E_1] := E_2, \mathsf{tq} = (E_1 \mapsto E_2), \mathcal{R} = [\![\mathbf{Emp}]\!], \mathcal{G} = [\![\mathbf{Emp}]\!]$。我们可以对 $\mathfrak{n}$ 进行归纳假设:

- Base case: $\mathfrak{n} = 0$. 这种情形平凡可证。
- Suppose $\mathfrak{n} = \mathfrak{j} + 1$. 根据定义5.5.4我们有四个子目标：
  (a) $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$;
  直接根据前提可证。
  (b) 若 $\mathrm{C} = \mathbf{skip}$, 则 $\Delta \models_{\mathrm{I}} tq$;
  由于 $\mathrm{C} \neq \mathbf{skip}$，所以平凡可证。
  (c) 若 $\mathrm{C} \neq \mathbf{skip}$, 则存在 C' 并且 Δ' 使得 $\mathrm{I} \vdash (\mathrm{C}, \Delta) \rightsquigarrow (\mathrm{C}', \Delta')$, $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $\mathcal{R}, \mathrm{I} \models_{\mathfrak{j}} (\mathrm{C}', \Delta') : (\mathcal{G}, tq)$;
  为了证明，我们对 C' 进行分析:

– $C' = C$. 这意味着我们有 $I \vdash (C, \Delta) \rightsquigarrow (C, \Delta')$。根据图 5.7中的操作语义, 我们知道这一步状态转换是 “Write from write buffer to memory with ownership transfer”。因此我们有 $\Delta \rhd_I \Delta'$ 并且 $\Delta.buf \neq \textbf{nil}$。由于我们有 $\mathcal{R} \models_I \Delta : \mathcal{G}$, 我们可以运用推论5.5.3 得到存在 Δ' 使得 $\Delta \rhd_I \Delta''$, $(\Delta.m_s, \Delta''.m_s) \in \mathcal{G}$, 并且 $\mathcal{R} \models_I \Delta'' : \mathcal{G}$。运用推论5.5.5, 我们有 $\Delta' = \Delta''$。所以我们知道 $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$ 并且 $\mathcal{R} \models_I \Delta' : \mathcal{G}$。运用推论5.5.6, 我们有 $\Delta' \models_I (E_1 \mapsto \_)$。根据归纳假设, 我们有 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$。所以我们能证明该情况。

– $C' = \textbf{skip}$. 意味着 $I \vdash (C, \Delta) \rightsquigarrow (\textbf{skip}, \Delta')$. 根据图 5.7中的操作语义, 我们知道这一步状态转移是 “Direct Write to memory”。因此我们有 $\Delta' = (m_s, m_l[a \mapsto v], rf, buf)$，其中 $a = [\![E_1]\!], v = [\![E_2]\!]$. 因为 $\Delta \models_I (E_1 \mapsto \_)$, 我们有 $\Delta' \models_I (E_1 \mapsto E_2)$。因此我们知道 $\Delta.buf = \Delta.buf' = \textbf{nil}$。由于 $\Delta.m_s = \Delta'.m_s$, 我们可以有 $(\Delta.m_s, \Delta'.m_s') \in \mathcal{G}$. 因为 $I = \textbf{emp}$ 并且 $\Delta = \blacktriangleright_I \Delta$, 我们知道 $[\![E_1]\!] \in \mathsf{dom}(\blacktriangleright_I \Delta.m_l)$。我们还知道 $\Delta'.m_l = m_l[a \mapsto v]$。所以我们可以运用引理5.5.3得知 $\mathcal{R} \models_I \Delta' : \mathcal{G}$。最后，我们可以用在证明 Skip 规则中用的证明 (Case. 1) 来得到结论 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$。

(d) 对于任意的 m' 使得 $(\Delta.m_s, m') \in \mathcal{R}$, 我们有 $\mathcal{R}, I \models_j (C, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tq)$; 由于我们知道 $m_s = m_s' = \emptyset$，所以这种情况平凡可证。

3.

$$\frac{(\blacktriangleright tp) * Q \Longrightarrow (E_1 \mapsto \_) * P \qquad (\blacktriangleright tp) \ltimes ((E_1 \mapsto E_2) * P) \Longrightarrow G}{[I], G, I \vdash \{tp * Q\}[E_1] := E_2\{(E_1 \multimap E_2) \triangleright (tp * Q)\}} \text{ (S-ASSN)}$$

根据定义5.5.3, 我们需要证明 Δ, 若 $\Delta \models_I tp$ 并且 $\mathcal{R} \models_I \Delta : \mathcal{G}$, 则 $\mathcal{R}, I \models_n (C, \Delta) : (\mathcal{G}, tq)$ 对于任意的 n 成立, 其中 $C = [E_1] := E_2$, $\mathcal{R} = [\![I]\!]$,$\mathcal{G} = [\![G]\!]$。我们对 n 进行归纳假设:

- Base case: $n = 0$. 平凡可证。
- Suppose $n = j + 1$. 展开定义5.5.4，我们有 4 个子目标。

(a) $\mathcal{R} \models_I \Delta : \mathcal{G}$;
根据前提可证。

(b) 若 $C = \textbf{skip}$, 则 $\Delta \models_I tq$;
由于 $C \neq \textbf{skip}$，故该情形平凡可证。

(c) 若 $C \neq \textbf{skip}$, 则存在 C' 并且 Δ' 使得 $I \vdash (C, \Delta) \rightsquigarrow (C', \Delta')$, $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 以及 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$;
为了证明，我们对 C' 进行分析:

– $C' = C$。意味着 $I \vdash (C, \Delta) \rightsquigarrow (C, \Delta')$。根据图 5.7中的操作语义, 我们知道 “Write from write buffer to memory with ownership transfer”。因此我们有 $\Delta \rhd_I \Delta'$ 并且 $\Delta.buf \neq \textbf{nil}$. 由于我们有 $\mathcal{R} \models_I \Delta : \mathcal{G}$, 我们可以运用推论5.5.3 得到存在 Δ' 使得 $\Delta \rhd_I \Delta''$, $(\Delta.m_s, \Delta''.m_s) \in \mathcal{G}$, 并且 $\mathcal{R} \models_I \Delta'' : \mathcal{G}$。运用推论5.5.5, 我们有 $\Delta' = \Delta''$。所以我们知道 $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$ 并且 $\mathcal{R} \models_I \Delta' : \mathcal{G}$。运用推论5.5.6, 我们有 $\Delta' \models_I tp$。根据归纳假设, 我们知道有 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$. 现在我们就能证明该情形。

– $C' = \textbf{skip}$。意味着我们有 $I \vdash (C,\Delta) \rightsquigarrow (\textbf{skip},\Delta')$。假设 $[\![E_1]\!] = a$ 并且 $[\![E_2]\!] = v$。根据图 5.7中的操作语义, 我们知道这一步状态转换是 “Write to write buffer”。因此我们有 $a \in \mathsf{dom}((\blacktriangleright_I \Delta).m_s)$, $\Delta.m_s = \Delta'.m_s$, $\Delta.m_l = \Delta'.m_l$ 并且 $\Delta'.buf = (a,v) :: (\Delta.buf)$。显然，我们有 $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$ 并且 $\Delta' \models_I tq$。下一步我们需要证明 $\mathcal{R} \models_I \Delta' : \mathcal{G}$。由于 $R = [I]$, 意味着环境不会对共享内存产生影响。因此我们有 $\mathsf{sta}(tp * Q, R)$。我们还知道 $\Delta \models_I tp * Q$。则对于任意的 $(m_{s1}, m_{l1}, rf_1, \textbf{nil}) \in (\blacktriangleright_I^{\mathcal{R}} \Delta)$, 我们知道
$(m_{s1}, m_{l1}, rf_1, \textbf{nil}) \models_I \blacktriangleright tp * Q$。由于 $(\blacktriangleright tp) \ltimes ((E_1 \mapsto E_2) * P)$, 我们知道 $(m_{s1}, m_{s1}[a \mapsto v]) \in \mathcal{G}$。由于我们有 $\mathcal{R} \models_I \Delta : \mathcal{G}$, 并且 $\Delta'.buf = (a,v) :: \Delta.buf$，所以我们可以运用引理5.5.4 来证明 $\mathcal{R} \models_I \Delta' : \mathcal{G}$。
所以我们有 $\mathcal{R} \models_I \Delta' : \mathcal{G}$ 并且 $\Delta' \models_I tq$, 则我们可以运用 Case. 1的情况来得到 $\mathcal{R}, I \models_j (\textbf{skip}, \Delta') : (\mathcal{G}, tq)$。最终我们可以证明这一情形。

(d) 对于任意的 m' 使得 $(\Delta.m_s, m') \in \mathcal{R}$, 我们有
$\mathcal{R}, I \models_j (C, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tq)$;
由于 $R = [I]$, 我们有 $\mathsf{sta}(tp{*}Q, R{*}\mathsf{Id})$。因此对于任意的 $(\Delta.m_s, m') \in \mathcal{R}$, 有 $m' = \Delta.m_s$。所以我们知道 $(m', \Delta.m_l, \Delta.rf, \Delta.buf) \models_I tp * Q$. 因此我们可以根据归纳假设进行证明。

4.

$$\frac{R, G, I \vdash \{tp\} C_1 \{tr\} \qquad R, G, I \vdash \{tr\} C_2 \{tq\}}{R, G, I \vdash \{tp\} C_1; C_2 \{tq\}} \text{ (SEQ)}$$

根据定义5.5.5, 我们需要证明对于任意的 Δ, 若 $\Delta \models_I tp$ 并且 $\mathcal{R} \models_I \Delta : \mathcal{G}$, 则 $\mathcal{R}, I \models_n (C, \Delta) : (\mathcal{G}, tq)$ 对于任意的 n 成立，其中 $C = C_1; C_2$。我们可以对 n 进行归纳假设:

- Base case: $n = 0$. 平凡可证。
- Suppose $n = j + 1$. 根据定义5.5.4我们有 4 个子目标：
  (a) $\mathcal{R} \models_I \Delta : \mathcal{G}$;
  根据前提可证。
  (b) 若 $C = \textbf{skip}$, 则 $\Delta \models_I tq$;
  由于 $C = C_1; C_2$, 我们有 $C_1 = \textbf{skip}$ and $C_2 = \textbf{skip}$. 因此我们可以知道 $tq = tp$ 并且证明该情形。
  (c) 若 $C \neq \textbf{skip}$, 则存在 C' 并且 Δ' 使得 $I \vdash (C,\Delta) \rightsquigarrow (C',\Delta')$, $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$;
  由于 $R, G, I \vdash \{tp\} C_1 \{tr\}$, 根据归纳假设我们有 $\mathcal{R}, \mathcal{G}, I \models \{tp\}\, C_1\, \{tr\}$。根据定义5.5.3, 我们知道 $\mathcal{R}, I \models_n (C_1, \Delta) : (\mathcal{G}, tr)$ 对于任意 n 成立。同理，我们有 $\mathcal{R}, I \models_n (C_2, \Delta_2) : (\mathcal{G}, tq)$ 对于任意 n 成立并且存在 Δ_2 使得 $\Delta_2 \models_I tr$。假设 $C' = C_1'; C_2$(若 $\mathsf{cmd}_1 = \textbf{skip}$, 假设 $C' = C_2'$. 证明过程是一样的)。根据操作语义我们有 $I \vdash (C_1, \Delta) \rightsquigarrow (C_1', \Delta')$。所以我们能够运用引理5.5.5 来得到 $\mathcal{R}, I \models_n (C_1', \Delta') : (\mathcal{G}, tq)$ 对于任意的 n 成立。使用归纳假设我们有 $\mathcal{R}, I \models_j (C', \Delta') : (\mathcal{G}, tq)$。
  (d) 对于任意的 m' 使得 $(\Delta.m_s, m') \in \mathcal{R}$, 我们有
  $\mathcal{R}, I \models_n (C, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tq)$;

由于我们有 $\mathsf{sta}(tp), (\Delta.m_s, m') \in \mathcal{R}$ 和 $\Delta \models_{\mathrm{I}} tp$, 我们知道 $(m', \Delta.m_l, \Delta.rf, \Delta.buf) \models_{\mathrm{I}} tp$。因此我们可以根据归纳假设证明结论。

5.

$$\frac{[\mathrm{I}], G, \mathrm{I} \vdash \{tp\}\iota\{tq\} \qquad \mathsf{sta}(\{tp, tq\}, R * \mathsf{Id}) \qquad \mathrm{I} \diamond \{R, G\}}{R, G, \mathrm{I} \vdash \{tp\}\iota\{tq\}} \text{ (ATOM-R)}$$

运用推论5.5.7可得结论。

6.

$$\frac{\begin{array}{c} R, G, \mathrm{I} \vdash \{tp\}\mathrm{C}\{tq\} \qquad \mathsf{sta}(tr, R' * \mathsf{Id}) \qquad \mathrm{I}' \diamond \{R', G'\} \\ tr \Longrightarrow \mathrm{I}' * \mathsf{true} \qquad tr \Rrightarrow G' * \mathbf{true} \end{array}}{R * R', G * G', \mathrm{I} * \mathrm{I}' \vdash \{tp * tr\}\mathrm{C}\{tq * tr\}} \text{ (FRAME)}$$

根据定义5.5.5, 我们需要证明对于任意的 Δ, 若 $\Delta \models_{\mathrm{I}*\mathrm{I}'} tp * tr$ 并且 $\mathcal{R} \models_{\mathrm{I}*\mathrm{I}'} \Delta : \mathcal{G}$, 则 $\mathcal{R}, \mathrm{I} * \mathrm{I}' \models_{\mathfrak{n}} (\mathrm{C}, \Delta) : (\mathcal{G}, tq * tr)$ 对于任意的 $\mathfrak{n}$ 成立，其中 $\mathcal{R} = [\![R * R']\!], \mathcal{G} = [\![G * G']\!]$. 我们可以对 $\mathfrak{n}$ 进行归纳假设:

- Base case: $\mathfrak{n} = 0$. 平凡可证。
- Suppose $\mathfrak{n} = \mathfrak{j} + 1$. 根据定义5.5.4我们有 4 个子目标：
  (a) $\mathcal{R} \models_{\mathrm{I}} \Delta : \mathcal{G}$;
  根据前提可证。
  (b) 若 $\mathrm{C} = \mathbf{skip}$, 则 $\Delta \models_{\mathrm{I}} tq$; 这个情况平凡可证。
  (c) 若 $\mathrm{C} \neq \mathbf{skip}$, 则存在 C' 以及 Δ' 使得 $\mathrm{I} * \mathrm{I}' \vdash (\mathrm{C}, \Delta) \rightsquigarrow (\mathrm{C}', \Delta')$, $(\Delta.m_s, \Delta'.m_s) \in \mathcal{G}$, 并且 $\mathcal{R}, \mathrm{I} * \mathrm{I}' \models_{\mathfrak{j}} (\mathrm{C}', \Delta') : (\mathcal{G}, tq * tr)$;
  我们能够找到 Δ_1 和 Δ_2 使得 $\Delta = \Delta_1 * \Delta_2$ 以及 $\Delta_2 \models_{\mathrm{I}'} tr$
  根据前提我们知道，存在 C' 以及 Δ_1' 使得 $\mathrm{I} \vdash (\mathrm{C}, \Delta_1) \rightsquigarrow (\mathrm{C}', \Delta_1')$, $(\Delta_1.m_s, \Delta_1'.m_s) \in \mathcal{G}_1$, 并且 $\mathcal{R}_1, \mathrm{I} \models_{\mathfrak{j}} (\mathrm{C}', \Delta_1') : (\mathcal{G}_1, tq)$，其中 $\mathcal{G}_1 = [\![R]\!]$ 以及 $\mathcal{R} = [\![G]\!]$。
  由于 C 不会改变 tr 所表示的内存区域，我们容易知道 $\Delta' = \Delta_1' \uplus \Delta_2$。由 $tr \Rrightarrow G' * \mathbf{true}$ 的定义展开我们知道，$(\Delta_1.m_s, \Delta_1'.m_s) \in \mathcal{G}_1$。根据归纳假设，我们可以知道 $\mathcal{R}, \mathrm{I} * \mathrm{I}' \models_{\mathfrak{j}} (\mathrm{C}', \Delta') : (\mathcal{G}, tq * tr)$。因此我们可以证明该情况。
  (d) 对于任意的 m' 使得 $(\Delta.m_s, m') \in \mathcal{R}$, 我们有
  $\mathcal{R}, \mathrm{I} * \mathrm{I}' \models_{\mathfrak{n}} (\mathrm{C}, (m', \Delta.m_l, \Delta.rf, \Delta.buf)) : (\mathcal{G}, tq * tr)$;
  我们能够找到 Δ_1 以及 Δ_2 使得 $\Delta = \Delta_1 \uplus \Delta_2$ 并且 $\Delta_2 \models_{\mathrm{I}'} tr$。根据归纳假设，我们有对于任意的 m' 使得 $(\Delta_1.m_s, m_1') \in \mathcal{R}_1$, 我们有 $\mathcal{R}_1, \mathrm{I} \models_{\mathfrak{n}} (\mathrm{C}, (m_1', \Delta_1.m_l, \Delta_1.rf, \Delta_1.buf)) : (\mathcal{G}_1, tq)$, 其中 $\mathcal{G}_1 = [\![R]\!]$ 以及 $\mathcal{R} = [\![G]\!]$。
  我们还知道根据 $\mathsf{sta}(tr, R' * \mathsf{Id})$, 可以有 $\mathsf{sta}(tp * tr, R * R')$。因此我们可以得出结论。

□

致　谢

现在是 2015 年 4 月 19 日凌晨，我刚敲下博士论文的最后一个字，深夜寂静，然而落笔却感概万千。读博士期间大抵最容易丢失三样东西：一份坚持而美好的爱情，一颗对生活积极乐观的心，以及，一头茂密的乌发。幸运的是我仅丢失了最后一样就完成了学业。

首先，我要感谢我的导师，冯新宇教授。冯老师的严谨的治学之道以及学术问题上的经验和智慧，总是让我惊叹不已。感谢他不厌其烦地指导我，帮助我，培养我严密的逻辑思维，也很抱歉自己在过去的几年中常常让冯老师失望。良师益友，冯老师不仅仅是授之以鱼，而且还授之以渔，我从他身上不仅仅是学习到这个方向的专业知识，更加学习到如何以做好科研的态度去做好一件事情，我相信这会是往后可能充满变数的雾气寥寥中生活中那最坚定不移的指路灯。

感谢陈意云教授在我读博期间的敦敦教诲。感谢邵中教授，让我能够有机会作为访问学生在耶鲁度过受益匪浅的一年。

感谢百科全书般的蒋新宇，会使一百零八样编程语言的庄重，什么学术问题都懂的梁鸿谨，写操作系统内核跟写 Hello,World 一样容易的张昊中，以及各位已经毕业和还在奋斗的实验室同学，感谢你们的陪伴。

感谢中国科学技术大学，作为学生我在这里度过了整整十年。感谢这一路上来来往往的兄弟姐妹，在最宝贵的年华里，是你们伴随着我的成长。人生不相见，动如参与商。或许我们都在各自路上渐行渐远，但和你们一起度过的快乐时光永远是我心中最美好的部分。

谢谢我的女朋友在过去的若干年里对我的无微不至的照顾，你是我前行最大的动力。

谢谢我的家人，谢谢你们一直以来的支持和鼓励, 你们是我完成学业坚强的后盾。

最后, 谢谢爸爸妈妈，我爱你们。

张扬

2015 年 4 月 19 日

在读期间发表的学术论文与取得的研究成果

研究工作：

1. **2009-2010, 验证条件生成器的构建器**: 我开发了一个用于自动化构建验证条件生成器的工具。这个工具本质上非常接近 yacc, 也是一个 LALR 语法分析器的产生器，通过在输入中规范程序语言文、语法以及语义，自动化构建出一个能够分析该语言并产生相应验证条件的生成器。

2. **2010-2013,Operational Happens-Before 内存模型**: 我们提出了一个 Happens-before 内存模型（HMM）的变种，我们希望能将它作为类 Java 语言的内存模型基础。我们的模型通过给出操作语义来规范程序语言在抽象机上的行为来模拟 HMM 并且可以对很多优化和预测算法进行抽象。

3. **2013-2014,CertiKOS 项目**：我参与了耶鲁大学 Flint 小组 Certikos 的代码开发工作。这一年中我的主要工作是对进程间通信模块（inter-process communication）进行开发和优化，以及虚拟机管理和 Bootloader 开发等。

4. **2013-2015,TSO 模型程序逻辑**我们设计了一个用于在 TSO 内存模型上验证并发算法的程序逻辑。TSO 内存模型广泛运用于 X86 和 SPARC_TSO 处理器家族中，并且也被提议作为一些高级语言的内存模型。我们使用我们开发的逻辑验证了一些有代表性的算法在 TSO 上的正确性，包括 Peterson 锁算法，Treibier 无锁栈算法，并发 GCD 算法以及 Spinlock 的优化实现算法。

已发表论文：

1. Yang Zhang,Xinyu Feng, An Operational Approach to Happens-Before Memory Model. Seventh International Symposium on Theoretical Aspects of Software Engineering (TASE), 2013 : 121-128, hold in Birmingham, UK, July 1-3,2013.

2. Yang Zhang,Xinyu Feng, An Operational Approach to Happens-Before Memory Model(Extended Version). Accepted by Frontier of Computer Science.

3. Zhaopeng Li, Yang Zhang, Yiyun Chen, A Method to Generate Verification Condition Generator. Fifth International Symposium on Theoretical Aspects of Software Engineering (TASE), 2011:239-242, hold in Xi’an China, Aug 29-31,2011.

待发表论文：

1. Yang Zhang, Xinyu Feng, Program Logic for Local Reasoning in TSO.