软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):2896−2917 [doi: 10.13328/j.cnki.jos.006604]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

TSO 内存模型下限界可线性化的可判定性研究*

王 超[1], 吕 毅[2,3], 吴 鹏[2,3], 贾巧雯[2,3]

[1](西南大学 计算机与信息科学学院 软件研究与创新中心, 重庆 400715)

[2](计算机科学国家重点实验室(中国科学院 软件研究所), 北京 100190)

[3](中国科学院大学, 北京 100049)

通信作者: 王超, E-mail: wangch1@swu.edu.cn; 吕毅, E-mail: lvyi@ios.ac.cn

摘 要: TSO-to-TSO 可线性化、TSO-to-SC 可线性化和 TSO 可线性化是 Total Store Order (TSO)内存模型下可线性化的 3 个变种. 提出了 *k*-限界 TSO-to-TSO 可线性化和 *k*-限界 TSO 可线性化, 考察了 *k*-限界 TSO-to-TSO 可线性化、*k*-限界 TSO-to-SC 可线性化和 *k*-限界 TSO 可线性化的验证问题. 它们分别是这 3 种可线性化的限界版本, 都使用 *k*-扩展历史, 这样的扩展历史对应的执行有着限界数目(不超过 *k* 个)的函数调用、函数返回、调用刷出和返回刷出动作. *k*-扩展历史对应执行中的写动作数目是不限界的, 进而执行中使用的存储缓冲区的大小也是不限界的, 对应的操作语义是无穷状态迁移系统, 所以 3 个限界版本可线性化的验证问题是不平凡的. 将定义在并发数据结构与顺序规约之间的 *k*-限界 TSO-to-TSO 可线性化、*k*-限界 TSO-to-SC 可线性化和 *k*-限界 TSO 可线性化的验证问题归约到 *k*-扩展历史集合之间的 TSO-to-TSO 可线性化问题, 从而以统一的方式验证了 TSO 内存模型下可线性化的 3 个限界版本. 验证方法的关键步骤是判定一个并发数据结构是否有一个特定的 *k*-扩展历史. 证明了这个问题是可判定的, 证明方法是将这一问题归约为已知可判定的易失通道机器的控制状态可达问题. 本质上, 这一归约将每一个函数调用或函数返回动作转化为写、刷出或 *cas* (compare-and-swap)动作. 在 TSO-to-TSO 可线性化的定义中, 一个函数调用或函数返回动作会同时影响存储缓冲区和控制状态. 为了模拟函数调用或函数返回动作对存储缓冲区的影响, 在每个函数调用或函数返回动作之后立刻执行一个特定的写动作. 这个写动作及其对应的刷出动作模拟了函数调用或函数返回动作对存储缓冲区的影响. 引入观察者进程, 为每个函数调用或函数返回动作“绑定”一个观察者进程的 *cas* 动作, 以这种方式模拟了函数调用或函数返回动作对控制状态的影响. 因此证明了 TSO 内存模型下可线性化的这 3 个限界版本都是可判定的, 进而证明了在 TSO 内存模型下判定可线性化的这 3 个限界版本的复杂度都在递归函数的 Fast-Growing 层级 $\mathfrak{F}_{\omega^\omega}$ 中. 通过证明已知对应复杂度的单通道简单通道机器的可达问题和 TSO 内存模型下可线性化的 3 个限界版本可以互相归约得到这个结论.

关键词: 并发数据结构; 可线性化; TSO 内存模型; 可判定性; 易失通道机器

中图法分类号: TP311

中文引用格式: 王超, 吕毅, 吴鹏, 贾巧雯. TSO 内存模型下限界可线性化的可判定性研究. 软件学报, 2022, 33(8): 2896−2917. http://www.jos.org.cn/1000-9825/6604.htm

英文引用格式: Wang C, Lü Y, Wu P, Jia QW. Decidability of Bounded Linearizability on TSO Memory Model. Ruan Jian Xue Bao/ Journal of Software, 2022, 33(8): 2896−2917 (in Chinese). http://www.jos.org.cn/1000-9825/6604.htm

* 基金项目: 国家自然科学基金(62002298, 62072443); 中央高校基本科研业务费专项资金(SWU019036); 中国科学院对外合作重点项目(GJHZ1844)

本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.

收稿时间: 2021-09-05; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

Decidability of Bounded Linearizability on TSO Memory Model

WANG Chao[1], LÜ Yi[2,3], WU Peng[2,3], JIA Qiao-Wen[2,3]

[1](Centre for Research and Innovation in Software Engineering, College of Computer and Information Science, Southwest University, Chongqing 400715, China)

[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: TSO-to-TSO linearizability, TSO-to-SC linearizability, and TSO linearizability are three typical variants of linearizability on the total store order (TSO) memory model. This study proposes *k*-bounded TSO-to-TSO linearizability and *k*-bounded TSO linearizability, and investigates the verification problems of *k*-bounded TSO-to-TSO linearizability, *k*-bounded TSO-to-SC linearizability, and *k*-bounded TSO linearizability that are bounded versions of the above variants of linearizability, defined on *k*-extended histories. Although the corresponding executions of *k*-extended histories contain a bounded number (less or equal than *k*) of call, return, flushCall and flushReturn actions, the verification problems of these three bounded versions of linearizability are non-trivial since the corresponding executions of *k*-extended histories may still contain an unbounded number of write actions and use store buffers with an unbounded capacity, which makes their operational semantics built upon infinite state transition systems. The *k*-bounded TSO-to-TSO linearizability problem, *k*-bounded TSO-to-SC linearizability problem, and *k*-bounded TSO linearizability problem between concurrent data structures and their sequential specifications are reduced into TSO-to-TSO linearizability problem between sets of *k*-extended histories, which provides a uniform framework for verifying the three bounded versions of linearizability on the TSO memory model. The key point of the proposed approach is to check if a concurrent data structure contains a specific *k*-extended history. It is proved that this problem is decidable by reducing it into the control state reachability problem of lossy channel machines, which is known decidable. This reduction essentially requires call and return actions to be transformed into write, flush or *cas* (compare-and-swap) actions. In the definition of TSO-to-TSO linearizability, a call or return action taken by a process changes the store buffer and the control state of the process at the same time. A specific write action is added immediately after each call or return action; thus, the influence on store buffers is mimicked by these specific write actions and their corresponding flush actions. To mimic the influence on control states, an observer process and bind specific *cas* actions of the observer process are introduced to each call or return actions. In this way, three bounded versions of linearizability are decidable on the TSO memory model are proved. Three bounded versions of linearizability on the TSO memory model are at level $\mathfrak{F}_{\omega^\omega}$ are further proved in the fast-growing hierarchy of recursive functions. This is proved by stating that the reachability problem of single-channel basic lossy channel machines, which is known in such complexity class, can be reduced into the three bounded versions of linearizability problems on the TSO memory model, while the latter problem can also be reduced into the former problem.

Key words: concurrent data structures; linearizability; TSO memory model; decidability; lossy channel machines

1 引 论

高效实现的并发数据结构能够充分利用多核系统的潜力, 提升并发程序的性能. 并发数据结构的抽象行为往往以顺序规约的形式给出, 正确性条件定义了并发数据结构是否符合顺序规约. 并发数据结构领域实际的正确性条件是可线性化(linearizability)[1]. 如果一个并发数据结构中的每个函数的执行都等效于在其调用和返回期间的某个时间点的瞬时执行, 那么该并发数据结构满足可线性化.

已有的可线性化验证的研究大部分假定使用顺序一致性(sequential consistency, SC)内存模型[2]. 在 SC 内存模型下, 并发程序的执行效果等同于所有进程对内存的访问以某种串行序交织进行, 这个串行序保留了每个进程访问内存动作的序关系, 并且所有对内存的写动作会立即被全局所见. 然而, 当代的多核处理器(如 x86/TSO[3], POWER[4]和 ARM[5])和编程语言(如 C/C++[6]和 Java[7])并不遵循 SC 内存模型, 而是遵循松弛内存模型(relaxed memory model). 为了提升性能, 松弛内存模型相比 SC 内存模型允许更多的行为. 以 TSO(total store order)内存模型[3]为例, 它为每个处理器关联了一个先入先出的存储缓冲区(store buffer), 使得进程对内存的写动作不再立即被全局所见. 尽管在每个实现 TSO 内存模型的多处理器系统中存储缓冲区的大小是固定的, 为了描述 TSO 内存模型下任意并发数据结构的实现, 文献中提出的 TSO 内存模型[8−11]往往为每个处理器关联一个大小不限界的先入先出存储缓冲区; 否则, 使用固定大小存储缓冲区的 TSO 内存模型总是无法建模需要更大存储缓冲区的 TSO 内存模型实现. 在 TSO 内存模型中, 在执行写动作时, 处理器将待更改的内存单

元及其新值先放入对应的存储缓冲区中，然后在将来某个不确定时刻将缓存的写动作从存储缓冲区中刷出(flush)，并修改对应的内存单元.

之前的文献已经提出了若干可线性化在松弛内存模型下的变种，例如 TSO 内存模型下的 TSO-to-TSO 可线性化[11]、TSO-to-SC 可线性化[12]和 TSO 可线性化[13]、C++11 内存模型下的两个可线性化变种[14]以及 C11 内存模型下的一个可线性化变种[15]. TSO-to-TSO 可线性化要求当函数调用或函数返回发生时，进程额外向存储缓冲区放入一个标记；当标记离开存储缓冲区时，会引发调用刷出(flushCall)或返回刷出(flushReturn)动作. TSO-to-TSO 可线性化使用扩展历史，即函数调用、函数返回、调用刷出和返回刷出动作的序列，来表示并发数据结构的行为. 扩展历史记录了并发数据结构和客户程序以及存储缓冲区的交互. TSO-to-SC 可线性化只使用函数调用和函数返回动作的序列来表示并发数据结构的行为；TSO 可线性化本质上只使用函数调用和返回刷出动作的序列来表示并发数据结构的行为. TSO 内存模型下可线性化的这 3 个变种都可理解为：在保留特定动作之间的序的前提下，一个并发数据结构的行为是否可以变换到另一个并发数据结构或者顺序规约的行为. 这 3 个变种存在一定的内在联系，TSO-to-SC 可线性化和 TSO 可线性化关注的动作种类是 TSO-to-TSO 可线性化关注的动作种类的子集，TSO-to-SC 可线性化和 TSO 可线性化定义中需要保持的序也是 TSO-to-TSO 可线性化定义中需要保持的序的子集. 换句话说，TSO-to-TSO 可线性化在这 3 个变种中是最“难”的. TSO 可线性化和 TSO-to-TSO 可线性化也有着显著的区别：前者定义在并发数据结构和顺序规约之间，而后者定义在并发数据结构之间.

可线性化的验证问题有着固有的困难性. SC 内存模型下的可线性化问题一般是不可判定[10]，而在进程数目固定时是可判定的[16]. 迄今为止，松弛内存模型下可线性化的可判定性结果还很少. 我们已经证明了进程数目固定时，TSO-to-TSO 可线性化仍是不可判定的[17]，而 *k*-限界 TSO-to-SC 可线性化是可判定的[18]. 然而进程数目固定时，限界的 TSO-to-TSO 可线性化和限界的 TSO 可线性化的可判定性仍是未知的.

本文提出了 *k*-限界 TSO-to-TSO 可线性化和 *k*-限界 TSO 可线性化，它们是 TSO-to-TSO 可线性化和 TSO 可线性化的限界版本，只考虑限界数目的函数调用、函数返回、调用刷出和返回刷出动作序列. 需要注意的是，TSO 内存模型下对可线性化的这 3 个限界版本的验证问题仍然是非平凡的. 因为读、写、刷出和 *cas*(compare-and-swap)动作的数目是不限的，因此使用的存储缓冲区大小是无界的，这导致对应的操作语义是无穷状态迁移系统.

基于 TSO 内存模型下可线性化的 3 个变种间的联系，为了以统一的方式验证 TSO 内存模型下可线性化的 3 个限界版本，我们将 TSO 内存模型下两个并发数据结构是否符合 *k*-限界 TSO-to-TSO 可线性化和 *k*-限界 TSO-to-SC 可线性化以及并发数据结构和顺序规约之间是否符合 *k*-限界 TSO 可线性化的问题归约为验证两个限界长度扩展历史集合是否满足 TSO-to-TSO 可线性化的问题，从而以一种统一的方式对 TSO 内存模型下可线性化的这 3 个限界版本进行验证，并最终证明了可线性化的 3 个限界版本是可判定的.

具体来说，我们将验证两个并发数据结构的 *k*-限界 TSO-to-TSO 可线性化问题归约为验证两个并发数据结构的 *k*-扩展历史集合的 TSO-to-TSO 可线性化问题. 我们称长度不超过 *k* 的扩展历史为 *k*-扩展历史. 一个并发数据结构的 *k*-扩展历史集合，是指这个并发数据结构的执行对应的长度不超过 *k* 的扩展历史的集合. 我们需要计算 TSO 内存模型下一个并发数据结构的 *k*-扩展历史集合，计算方法如下：假定已经有了一种方法来判定 TSO 内存模型下一个并发数据结构是否有一条给定的 *k*-扩展历史，然后，由于可能的 *k*-扩展历史的数目是有限的，枚举列出所有可能的 *k*-扩展历史，并使用这个方法来判定每一条可能的 *k*-扩展历史是否属于这个并发数据结构. 我们将验证两个并发数据结构的 *k*-限界 TSO-to-SC 可线性化问题归约为两个只包含函数调用和函数返回动作的限界扩展历史集合的 TSO-to-TSO 可线性化问题. 我们将验证并发数据结构 *k*-限界 TSO 可线性化到正则顺序规约这个问题归约为若干个限界长扩展历史集合间的 TSO-to-TSO 可线性化问题.

两条 *k*-扩展历史是否符合 TSO-to-TSO 可线性化是可判定的，判定方法是：根据 TSO-to-TSO 可线性化的定义，枚举它们之间的一一映射函数并判断是否保留了特定动作之间的序. 两个 *k*-扩展历史集合是否符合 TSO-to-TSO 可线性化也是可判定的，判定方法为：对第 1 个集合中的每个扩展历史，寻找第 2 个集合中是否

有与其满足 TSO-to-TSO 可线性化关系的扩展历史. 因此, 我们给出了判定 TSO 内存模型下可线性化的 3 个限界版本的方法.

在上述验证过程中, 只剩下判定 TSO 内存模型下一个并发数据结构是否有一条给定的 *k*-扩展历史的方法未叙述, 现在给出判定方法. 受到文献[8]的启发, 我们希望将并发数据结构是否有一个特定的 *k*-扩展历史这个问题归约到易失通道机器(lossy channel machine)的控制状态可达问题, 而后者是可判定的[8]. 然而, 文献[8]中的归约方法在这里并不适用, 因为它对应的并发系统中只包含内部、读、写和 *cas* 动作, 而不包含函数调用和函数返回动作. TSO-to-TSO 可线性化中的一个函数调用或函数返回动作需要同时完成两个任务: 第 1 个任务是将一个函数调用或函数返回标记插入对应进程的存储缓冲区, 而第 2 个任务是修改对应进程的控制状态. 证明中的关键步骤是修改内存模型和并发系统的操作语义, 将原本的一个函数调用或函数返回动作转化为一对写动作和 *cas* 动作. 修改后的并发系统与原来的并发系统有着相同的扩展历史集合, 而且更易于使用归约的思想判定其是否有一条给定的 *k*-扩展历史.

对一个函数调用或函数返回动作的转化过程如下: 首先引入一个新的内存单元 z_f, 并将一个函数调用或函数返回动作分解为一个"纯粹"的函数调用或函数返回动作和一个对 z_f 的写动作. 前者只会修改进程的控制状态, 而后者写入一个函数调用标记或函数返回标记. 我们修改了并发系统的操作语义, 使得当 z_f 的写被刷出到内存时, 依据这个写的值是函数调用标记还是函数返回标记, 发出一个调用刷出或返回刷出动作. 因此, 在原 TSO-to-TSO 可线性化定义中的函数调用和函数返回动作的第 1 个任务被对 z_f 的写动作和对应的调用刷出和返回刷出动作所模拟.

可以使用文献[18]工作中的思想来处理这些"纯粹"的函数调用和函数返回动作. 我们需要每个"纯粹"的函数调用或函数返回动作都能够立刻被其他进程所见, 为此, 修改操作语义, 为每个函数调用和函数返回动作"绑定"一个特定的 *cas* 动作. 具体来说, 引入了一个新的内存单元 z_w 和一个额外的"观察者"进程, 后者以非确定猜测的方式持续地使用 *cas* 语句更新 z_w. 我们修改了内存模型, 使得观察者进程的每一个 *cas* 动作之后都立刻伴随一个函数调用或函数返回动作. 通过这些方法, 扩展历史中的函数调用和函数返回动作与 z_w 的 *cas* 动作相对应, 调用刷出和返回刷出动作与 z_f 的"刷出"动作相对应.

对并发数据结构的每一条扩展历史, 在修改后的操作语义中存在着一条有着相同扩展历史的执行, 且后者最终清空了每个进程的存储缓冲区, 我们称这样的执行为对应扩展历史的标记见证. TSO 内存模型下并发数据结构有一条特定的扩展历史, 当且仅当在修改后的操作语义上存在这个扩展历史的标记见证. 给定一条扩展历史 *eh*, 我们构造一个易失通道机器 M_i^{eh} 来模拟并发系统对应 *eh* 的执行中进程 *i* 的行为. M_i^{eh} 只有一个通道, 它既执行进程 *i* 的程序, 也非确定地猜测其他进程的动作, 并且在执行每个猜测的其他进程的写动作时也会向通道放入一个数据项. 检测在特定的状态之间是否存在扩展历史 *eh* 的标记见证问题, 就被归约为易失通道机器 $M_1^{eh-w},...,M_{n+1}^{eh-w}$ 的乘积上的控制状态可达问题. 易失通道机器 M_i^{eh-w} 是通过将易失通道机器 M_i^{eh} 中的 *cas* 迁移转化为写迁移, 再将所有非写迁移转化为内部迁移得到的. 由于 M_i^{eh-w} 保证了通道中不会丢失的数据数目有穷, 并且每个写迁移放入通道的数据项包含了整个内存的快照, 所以丢失一些通道内容不会影响可达, 这使得归约成立. 由于这样的特定状态对数目是有穷的, 能够把检测扩展历史的标记见证问题, 即 TSO 内存模型下特定扩展历史的存在问题, 归约到易失通道机器上的控制状态可达问题. 即判定了 TSO 内存模型下一个并发数据结构是否有一条给定的 *k*-扩展历史. 根据之前的讨论, 我们因此证明了 TSO 内存模型下可线性化的 3 个限界版本都是可判定的.

最后证明在 TSO 内存模型下判定可线性化的这 3 个限界版本的复杂度都在递归函数的 Fast-Growing 层级[19] $\mathfrak{F}_{\omega^\omega}$ 中. 证明方法是: 证明单通道简单通道机器的可达性这一已知复杂度的问题[20]和 TSO 内存模型下可线性化的这 3 个限界版本可以互相归约.

- 相关工作

SC 内存模型上的可线性化验证问题得到了学术界的广泛关注[10,16,21−25]. 在可判定性方面, Alur 等人[16]证明了进程数目固定时可线性化问题是可判定的, 而 Bouajjani 等人[10]证明了在一般情况下可线性化问题不可

判定. 在验证方面, 研究者们提出了多种验证可线性化的方法: Liang 等人[21,22]使用定理证明方法手工证明了一些典型的并发数据结构满足可线性化; Bouajjani 等人[23,24]将可线性化的验证归约到自动机的可达问题; Liu 等人[25]通过模型检测的方式验证并发数据结构是否满足可线性化.

关于 TSO 内存模型下可线性化的研究工作目前较少. 在可判定性方面, 文献[17]中证明了 TSO-to-TSO 可线性化问题在进程数目固定时已经不可判定, 文献[18]证明了 k-限界 TSO-to-SC 可线性化问题在进程数目固定时是可判定的. 在验证方面, Derrick 等人[26]和 Travkin 等人[27]使用定理证明器 KIV 验证了一些并发数据结构满足 TSO 内存模型下的可线性化.

Atig 对 TSO 内存模型下有穷状态并发程序的安全性和活性的已知可判定性问题进行了综述[28]. 一般而言, 并发数据结构的正确性和活性是在函数粒度, 基于函数调用和函数返回动作定义的, 而文献[28]中的安全性和活性关注的是并发程序的特定状态, 这两者的粒度不同. 文献[17,18]受到了 Atig 等人[8]工作的启发, 他们揭示了 TSO 内存模型下并发系统和易失通道机器的联系, 将并发系统的状态可达问题归约到易失通道机器的控制状态可达问题.

文献[18]考虑对 k-限界 TSO-to-SC 可线性化的验证, 每个函数调用或函数返回动作只修改进程的控制状态. 在本文中, 我们考虑的情况更加复杂, 每个函数调用或函数返回动作同时完成两个任务. 文献[18]中对 k-限界 TSO-to-SC 可线性化的结论是本文的一个推论.

2 并发系统

本节引入了并发数据结构、客户程序、最一般客户(most general client)和并发系统的概念, 并引入了并发系统的两个操作语义, 即文献[11]中定义的 TSO-to-TSO 可线性化时使用的操作语义以及在 SC 内存模型下的操作语义.

2.1 基本概念

我们用 $l=\alpha_1\cdot\alpha_2\cdot\ldots\cdot\alpha_k$ 表示字母表 Σ 上的一个有穷序列 l, 这里, $\cdot$是连接符号, 且每个元素 $\alpha_i(1\leqslant i\leqslant k)$都属于字母表 Σ. 令 $|l|$和 $l(i)$分别表示有穷序列 l 的长度和第 i 个元素, 即 $|l|=k$ 且 $l(i)=\alpha_i$. 令 $l\uparrow\Sigma$为有穷序列 l 在字母表 Σ 上的投影. 给定函数 f, 令 $f[x:y]$为这样的一个函数: 除了将 x 映射为 y 之外, 对其他元素的映射都和 f 一样. 令ε为空序列, _表示某个数据项而不关注其值.

一个标号迁移系统(labelled transition system, LTS)是一个四元组 $A=(Q,\Sigma,\rightarrow,q_0)$. 这里, Q 是状态(也称为格局)的集合, Σ是迁移标号的字母表, $\rightarrow\subseteq Q\times\Sigma\times Q$ 是迁移关系, 而 q_0 是初始状态. A 的一条路径是一个有穷迁移序列 $q_0\xrightarrow{\beta_1}q_1\xrightarrow{\beta_2}\ldots\xrightarrow{\beta_k}q_k$. 如果存在 A 的路径 $q_0\xrightarrow{\beta_1}q_1\xrightarrow{\beta_2}\ldots\xrightarrow{\beta_k}q_k$, 则称有穷序列 $t=\beta_1\cdot\beta_2\cdot\ldots\cdot\beta_k$ 是 A 的一条迹.

2.2 并发数据结构和客户程序

一个并发系统包含着并发运行的客户程序, 这些客户程序之间的交互是通过调用并发数据结构的函数来完成的. 一个并发数据结构提供若干函数以访问该数据结构. 为了简化符号, 假定每个函数有着一个参数并返回一个值. 并发数据结构和客户程序都可以有私有的内存.

令 X, M 和 D 是有穷的内存单元集合、有穷的函数名集合和有穷的数据域. 一个原语命令有着如下形式:

$$\tau|read(x,a)|write(x,a)|cas\_suc(x,a,b)|cas\_fail(x,a,b)|call(m,a)|return(m,a).$$

这里, $a,b\in D$, $x\in X$ 且 $m\in M$. 为了以标号迁移系统的方式定义并发数据结构和客户程序的语义, 我们在原语命令中包含了命令涉及的值. 例如, 读命令 $read(x,a)$中包含了读到的值 a. τ是对应内部动作的原语命令. 一个 *cas*(compare-and-swap)命令原子地执行一个读和(可能的)一个写动作. 一个成功执行的 *cas* 命令被写为 $cas\_suc(x,a,b)$, 它仅在 x 的值为 a 时执行, 并将 x 的值修改为 b. 一个失败的 *cas* 命令被写为 $cas\_fail(x,a,b)$, 它仅在 x 的值不为 a 时执行, 它的执行不会修改任何内存单元的值.

一个并发数据结构 L 被定义为一个五元组 $L=(X_L,M_L,D_L,Q_L,\rightarrow_L)$. 这里, X_L, M_L 和 D_L 分别是有穷的内存单元

集合、有穷的函数名集合和有穷数据域. $Q_L=\bigcup_{m\in M_L}Q_m$ 是各个函数 $m\in M_L$ 的程序位置集合的并集. 这里的一个程序位置存储了当前程序计数器以及当前进程的寄存器的值, 被视为一个状态. $\rightarrow_L=\bigcup_{m\in M_L}\rightarrow_m$ 是各个函数 $m\in M_L$ 的迁移关系的并集. 令 $PCom_L$ 是使用内存单元集合 X_L、函数名集合 M_L 和数据域 D_L 的(不包含函数调用和函数返回命令)原语命令的集合, 则$\rightarrow_m$ 定义为 $Q_m\times PCom_L\times Q_m$ 的一个子集. 任取 $m\in M_L$ 和 $a\in D_L$, 在 Q_m 中都存在一个特殊的"初始状态"$is_{(m,a)}$和"终止状态"$fs_{(m,a)}$: $is_{(m,a)}$表示函数 m 被调用且参数值为 a 时, 开始执行的第 1 个状态; $fs_{(m,a)}$表示函数 m 在执行返回且返回值为 a 之前执行的最后一个状态.

一个客户程序 C 被定义为一个五元组 $C=(X_C,M_C,D_C,Q_C,\rightarrow_C)$. 这里, X_C, M_C, D_C 和 Q_C 分别是有穷的内存单元集合、有穷的函数名集合、有穷的数据域和有穷的程序位置集合. 令 $PCom_C$ 是使用内存单元集合 X_C、函数名集合 M_C 和数据域 D_C 的原语命令集合, 则$\rightarrow_C$ 定义为 $Q_C\times PCom_C\times Q_C$ 的一个子集.

一个最一般客户是一个特殊的客户程序, 它不断以非确定的方式调用任意的函数并使用任意的参数. 最一般客户用来展示并发数据结构的所有可能行为. 一个最一般客户 MGC 被定义为一个五元组$(\varnothing,M_c,D_c,\{in_{clt},in_{lib}\},\rightarrow_{mgc})$. 这里, $\rightarrow_{mgc}=\{(in_{clt},call(m,a),in_{lib}),(in_{lib},return(m,a),in_{clt})|m\in M_c,a\in D_c\}$是迁移关系. in_{clt} 代表当前没有库函数正在执行, 而 in_{lib} 代表当前有库函数正在执行.

2.3 操作语义

让我们使用文献[11]中的思想来给出定义 TSO-to-TSO 可线性化时使用的操作语义. 假定并发系统 $C(L)$ 包含 n 个进程, 每个进程 $P_i(1\leqslant i\leqslant n)$运行一个客户程序 $C_i=(X_C,M_C,D_C,Q_C,\rightarrow_C)$, 而且所有的客户程序使用同一个并发数据结构 $L=(X_L,M_L,D_L,Q_L,\rightarrow_L)$. 这里的 C 是一个函数, 将每一个进程 P_i 映射到客户程序 C_i. 文献[11]中将 $C(L)$的操作语义定义为一个标号迁移系统$[\![C(L),n]\!]_{tt}=(Conf_{tt},\Sigma_{tt},\rightarrow_{tt},InitConf_{tt})$. 这里的下标 tt 表示 TSO-to-TSO 可线性化, 且 $Conf_{tt}$, Σ_{tt}, $\rightarrow_{tt}$ 和 $InitConf_{tt}$ 分别定义如下.

- 每个 $Conf_{tt}$ 中的格局是一个三元组(p,d,u), 其中, p 存储了每个进程的当前控制状态, d 存储了每个内存单元的取值, 而 u 存储了每个进程的存储缓冲区的内容;
- Σ_{tt} 是动作的集合, 包含如下元素:
  $\tau(i)|read(i,x,a)|write(i,x,a)|cas(i,x,a,b)|flush(i,x,b)|call(i,m,a)|return(i,m,a)|flushCall(i)|flushReturn(i)$.
  其中, $1\leqslant i\leqslant n$, $m\in M$, $x\in X_L\cup X_C$ 且 $a,b\in D$;
- 对每个进程 $P_i(1\leqslant i\leqslant n)$, 迁移关系$\rightarrow_{tt}$包含如下迁移: 读动作 $read(i,x,a)$需要确保或者当前进程存储缓冲区 $u(i)$中对应 x 的最新项是(x,a), 或者这样的项不存在且内存中 x 的值是 $a(d(x)=a)$; 写动作 $write(i,x,a)$并不直接修改内存, 而是将一个数据对(x,a)放入当前进程存储缓冲区 $u(i)$的尾部; 刷出动作 $flush(i,x,a)$从当前进程的存储缓冲区 $u(i)$的头部取出数据对(x,a), 并将内存中 x 的取值修改为 a; cas 动作 $cas(i,x,a,b)$会清空当前进程存储缓冲区 $u(i)$, 之后, 或者原子地确认内存中 x 的值 a 并将其修改为 b, 或者原子地确认内存中 x 的值不为 a 之后不修改任何内存; 函数调用动作 $call(i,m,a)$将一个标记 $call$ 放入当前进程存储缓冲区 $u(i)$的尾部, 并开始执行函数 m 在参数为 a 时的代码; 函数返回动作 $retun(i,m,a)$将一个标记 ret 放入当前进程存储缓冲区 $u(i)$的尾部并从函数 m 中返回, 返回值 a 并回到客户程序; 当标记 $call$ 或 ret 从进程的存储缓冲区 $u(i)$的头部刷出时, 会触发一个调用刷出动作 $flushCall(i)$或返回刷出动作 $flushReturn(i)$;
- 初始格局 $InitConf_{tt}$ 是一个三元组$(p_{init},d_{init},u_{init})$, 其中, p_{init} 将每个进程 ID 映射到一个初始状态, d_{init} 规定了 $X_L\cup X_C$ 中的内存单元的初始值, 而 u_{init} 将每个进程 ID 映射到一个空的存储缓冲区. 当每个进程都使用最一般客户时, 我们将此时的标号迁移系统$[\![C(L),n]\!]_{tt}$ 简写为$[\![L,n]\!]_{tt}$.

根据文献[12], 并发系统在 SC 内存模型上的操作语义定义为一个标号迁移系统$[\![C(L),n]\!]_{sc}=(Conf_{sc},\Sigma_{sc},\rightarrow_{sc},InitConf_{sc})$. $[\![C(L),n]\!]_{sc}$ 与$[\![C(L),n]\!]_{tt}$ 相似, 不同点是: 在 $Conf_{sc}$ 的每个格局中, 每个进程的存储缓冲区都固定为空; 每个写动作不再修改存储缓冲区而是直接修改内存; 函数调用和函数返回动作不再向存储缓冲区中放入标记, 因此也不再有调用刷出和返回刷出动作. 当每个进程都使用最一般客户时, 我们将此时的标号迁移系

统$[\![C(L),n]\!]_{sc}$简写为$[\![L,n]\!]_{sc}$. 在本文的技术报告版本[29]中, 可以找到$[\![C(L),n]\!]_{tt}$和$[\![C(L),n]\!]_{sc}$的详细定义.

3 正确性条件

在本节中, 我们引入 TSO-to-TSO 可线性化、(*k*-限界)TSO-to-SC 可线性化和 TSO 可线性化的定义, 并提出 *k*-限界 TSO-to-TSO 可线性化和 *k*-限界 TSO 可线性化的定义.

3.1 (限界)TSO-to-TSO可线性化

TSO-to-TSO 可线性化[11]是可线性化[1]在 TSO 内存模型下的一个变种. TSO-to-TSO 可线性化满足抽象定理[11,12,14], 后者是正确性条件的一个重要属性. 一个正确性条件满足抽象定理, 如果使用并发数据结构时的每个客户程序行为, 在将并发数据结构替换为抽象实现时仍会发生.

在 TSO 内存模型中, 数据结构和客户程序都可能通过存储缓冲区展示出副作用. 为了确保抽象定理成立, 除了函数调用和函数返回, 并发数据结构还需要记录如下两种时刻: 一个函数写进存储缓冲区的第 1 个数据项(如果存在), 从存储缓冲区中刷出到内存的时间点; 以及一个函数写进存储缓冲区的最后一个数据项(如果存在), 从存储缓冲区中刷出到内存的时间点. 这两种时间点分别对应了调用刷出和返回刷出动作. 令Σ_{cal}, Σ_{ret}, Σ_{fcal}和Σ_{fret}分别为函数调用动作、函数返回动作、调用刷出动作和返回刷出动作的集合. 一个扩展历史是一个字母表$(\Sigma_{cal}\cup\Sigma_{ret}\cup\Sigma_{fcal}\cup\Sigma_{fret})^*$上的有穷序列, 它用来表示并发数据结构的行为, 记录了数据结构和客户程序之间的交互以及数据结构和存储缓冲区之间的交互. 我们称 *eh* 为标号迁移系统 *A* 的一条扩展历史, 如果存在 *A* 的一条迹, 且其上的扩展历史为 *eh*. 令 *ehistory*(*A*)表示标号迁移系统 *A* 的扩展历史的集合, 令$eh|_i$为扩展历史 *eh* 在进程 P_i的动作集合上的投影.

给定两个扩展历史eh_1和eh_2, 如果对每个进程$P_i(1\leqslant i\leqslant n)$, $eh_1|_i=eh_2|_i$都成立, 那么称这两个扩展历史是等价的. 给定两个扩展历史eh_1和eh_2, 如果:

- eh_1和eh_2是等价的;
- 存在$\{1,\ldots,|eh_1|\}$和$\{1,\ldots,|eh_2|\}$之间的一一映射π, 使得对任意的$1\leqslant i\leqslant|eh_1|$, $eh_1(i)=eh_2(\pi(i))$都成立;
- 任取$1\leqslant i<j\leqslant|eh_1|$, 如果$eh_1(i)\in\Sigma_{ret}\cup\Sigma_{fret}$且$eh_1(j)\in\Sigma_{cal}\cup\Sigma_{fcal}$, 则$\pi(i)<\pi(j)$成立.

我们称eh_1是 TSO-to-TSO 可线性化到扩展历史eh_2的.

给定扩展历史的集合 S_1 和 S_2, 如果对每条扩展历史 $eh_1\in S_1$, 都存在着扩展历史 $eh_2\in S_2$, 使得 eh_1 是 TSO-to-TSO 可线性化到 eh_2 的, 则称 S_2 是 TSO-to-TSO 线性化了 S_1 的. 给定并发数据结构 L_1 和 L_2, 如果 *ehistory*($[\![L_2,n]\!]_{tt}$)是 TSO-to-TSO 线性化了 *ehistory*($[\![L_1,n]\!]_{tt}$)的, 称 L_2 在 n 进程下 TSO-to-TSO 线性化了 L_1.

一条 *k*-扩展历史是一条包含不超过 *k* 个动作的扩展历史. 给定数字 *k* 和标号迁移系统 *A*, 令 *k*-*ehistory*(*A*)表示标号迁移系统 *A* 上的 *k*-扩展历史集合. 我们提出了 *k*-限界 TSO-to-TSO 可线性化这个概念, 它只考虑 *k*-扩展历史, 是 TSO-to-TSO 可线性化的一个限界且可判定(在后面章节证明)的子类. 它的定义如下:

定义 1(*k*-限界 TSO-to-TSO 可线性化). 给定并发数据结构 L 和 L', 如果对每条 *k*-扩展历史 $eh\in k$-*ehistory*($[\![L,n]\!]_{tt}$), 都存在 *k*-扩展历史 $eh'\in k$-*ehistory*($[\![L',n]\!]_{tt}$), 使得 *eh* 是 TSO-to-TSO 可线性化到 eh'的, 那么我们称 L'在 n 进程下 *k*-限界 TSO-to-TSO 线性化了 L.

3.2 (限界)TSO-to-SC可线性化

TSO-to-SC 可线性化[12]是可线性化在 TSO 内存模型下的另一个变种, 它也满足抽象定理. 它关联了一个运行在 TSO 内存模型下的并发数据结构以及一个运行在 SC 内存模型上的数据结构的抽象实现. 一条历史是一个函数调用动作和函数返回动作的序列. TSO-to-SC 可线性化使用历史来表示并发数据结构的行为. 我们称 *h* 为标号迁移系统 *A* 的一条历史, 如果存在 *A* 的一条迹, 且其上的历史为 *h*. 令 *history*(*A*)为标号迁移系统 *A* 的历史的集合.

给定历史h_1和h_2, 如果:

- h_1和h_2是等价的;

- 存在{1,…,$|h_1|$}和{1,…,$|h_2|$}之间的一一映射π, 使得对任意的 $1\leqslant i\leqslant|h_1|$, $h_1(i)=h_2(\pi(i))$都成立;
- 取 $1\leqslant i<j\leqslant|eh_1|$, 如果 $eh_1(i)\in\Sigma_{ret}$且 $eh_1(j)\in\Sigma_{cal}$, 则$\pi(i)<\pi(j)$成立.

那么称 h_1 是 TSO-to-SC 可线性化到 h_2 的.

给定并发数据结构 L_1 和 L_2, 如果对每一条历史 $h_1\in history(\llbracket L_1,n\rrbracket_{tt})$, 存在历史 $h_2\in history(\llbracket L_2,n\rrbracket_{sc})$, 使得 h_1 是 TSO-to-SC 可线性化到 h_2 的, 那么称 L_2 在 n 进程下 TSO-to-SC 线性化了 L_1.

一条 k-历史是一条包含不超过 k 个动作的历史. 给定数字 k 和标号迁移系统 A, 令 k-$history(A)$表示标号迁移系统 A 上的 k-历史集合. 文献[18]提出了 k-限界 TSO-to-SC 可线性化, 它只考虑 k-历史, 是 TSO-to-SC 的一个限界且可判定[18]的子类. 它的定义如下:

定义 2(k-限界 TSO-to-SC 可线性化)[18]. 给定并发数据结构 L 和 L', 如果对每条 k-历史 $h\in k\text{-}ehistory(\llbracket L,n\rrbracket_{tt})$, 都存在 k-历史 $h'\in k\text{-}ehistory(\llbracket L',n\rrbracket_{sc})$, 使得 h 是 TSO-to-SC 可线性化到 h'的, 那么我们称 L'在 n 进程下 k-限界 TSO-to-SC 线性化了 L.

3.3 (限界)TSO可线性化

TSO 可线性化[13]是可线性化在 TSO 内存模型下的另一个变种. 和 TSO-to-TSO 可线性化以及 TSO-to-SC 可线性化不同的是, TSO 可线性化并不满足抽象定理. 给定函数返回动作 $return(i_1,m_1,a_1)$和函数调用动作 $call(i_2,m_2,a_2)$, 如果 $i_1=i_2$ 且 $m_1=m_2$, 则称这两个动作配对. 一条顺序历史是一条以函数调用开始, 并且除了最后一个动作之外, 每个函数调用之后紧接着配对的函数返回动作, 而每个函数返回动作之后紧接着函数调用动作的历史. TSO 可线性化将一个运行在 TSO 内存模型下的并发数据结构和其顺序规约关联起来, 后者是顺序历史的集合.

本质上, TSO 可线性化使用函数调用动作和调用刷出动作来表示并发数据结构的行为. 给定一个扩展历史 eh 及函数返回动作 $eh(j_1)=return(i,m,a)$和返回刷出动作 $eh(j_2)=flushReturn(i)$, 如果存在自然数 g, 使得在 $eh|_i$ 上, $eh(j_1)$是第 g 个函数返回动作, 而 $eh(j_2)$是第 g 个返回刷出动作, 则称这两个动作配对. 给定扩展历史 eh, 令 $Trans(eh)$是通过先删除函数返回和调用刷出动作, 再将返回刷出动作转化为配对的函数返回动作所得到的历史. $Trans(eh)$将一条扩展历史转化为一条“对应的历史”. TSO 可线性化使用通过 $Trans$ 转化扩展历史所得到的历史集合来表示并发数据结构的行为.

给定历史 h, 令 $complete(h)$为 h 的包含配对的函数调用动作和函数返回动作的最大子序列. 一条历史中的一个操作 e 包含一对函数调用动作 $inv(e)$以及之后配对的函数返回动作 $res(e)$. 一个顺序规约是一个对前缀封闭的顺序历史的集合. 给定扩展历史 eh 和顺序规约 $Spec$, 如果存在一条顺序历史 $s\in Spec$, 并且 $Trans(eh)$可以通过添加零个或若干个函数返回操作被扩展为一条历史 h', 使得:

- $complete(h')$和 s 是等价的;
- 对 $Trans(eh)$上的每一对操作 e_1 和 e_2, 如果在 $Trans(eh)$中, $res(e_1)$先于 $inv(e_2)$发生, 则在 s 中, $res(e_1)$也先于 $inv(e_2)$发生.

称 eh 是 TSO 可线性化到 $Spec$ 的.

给定并发数据结构 L 和顺序规约 $Spec$, 如果对任意扩展历史 $eh\in ehistory(\llbracket L,n\rrbracket_{tt})$, eh 都是 TSO 可线性化到顺序规约 $Spec$ 的, 则称 L 在 n 进程下 TSO 可线性化到顺序规约 $Spec$.

我们提出了 k-限界 TSO 可线性化这个概念, 它只考察限界的扩展历史, 是 TSO 可线性化的一个限界且可判定(在后面章节证明)的子类. 它的定义如下:

定义 3(k-限界 TSO 可线性化). 给定并发数据结构 L 和顺序规约 $Spec$, 如果对每条满足 $Trans(eh)\leqslant k$ 的扩展历史 $eh\in ehistory(\llbracket L,n\rrbracket_{tt})$, eh 都是 TSO 可线性化到 $Spec$ 的, 那么称 L 在 n 进程下 k-限界 TSO 可线性化到 $Spec$.

需要注意的是, 尽管 k-限界 TSO-to-TSO 可线性化、k-限界 TSO-to-SC 可线性化和 k-限界 TSO 可线性化只考虑限界数目的函数调用、函数返回、调用刷出和返回刷出动作, 但在每一对函数调用和函数返回之间可

能发生的动作数目是不限的, 进而使用的存储缓冲区大小是不限的. 因此, 对这 3 个 TSO 内存模型下可线性化的限界版本的验证需要对无穷状态标号迁移系统(操作语义)进行验证, 是非平凡的.

4 TSO 内存模型下可线性化的 3 个限界版本的验证思路

本节给出验证 TSO 内存模型下可线性化的 3 个限界版本(*k*-限界 TSO-to-TSO 可线性化、*k*-限界 TSO-to-SC 可线性化和 *k*-限界 TSO 可线性化)的验证思路. 我们将验证两个并发数据结构符合 *k*-限界 TSO-to- TSO 可线性化和 *k*-限界 TSO-to-SC 可线性化问题, 以及验证并发数据结构 *k*-限界 TSO 可线性化到正则的顺序规约这 3 个问题归约到验证两个 *k*-扩展历史集合符合 TSO-to-TSO 可线性化问题, 从而以一种统一的方法来完成对 TSO 内存模型下可线性化的 3 个限界版本的验证. 注意: 在考虑 *k*-限界 TSO 可线性化的可判定性问题时, 假定顺序规约是正则的.

- 统一的验证方法

我们通过如下两个步骤完成针对并发数据结构以及正则顺序规约之间的 *k*-限界 TSO-to-TSO 可线性化、*k*-限界 TSO-to-SC 可线性化和 *k*-限界 TSO 可线性化的验证.

(1) 在第 1 步, 构造出两个限界长度(限界的长度为 k 或 $k+n$)扩展历史集合 S_1 和 S_2. 由于函数名集合有穷、进程数目有穷且数据域有穷, 所有可能的 *k*-扩展历史集合和 *k*+*n*-扩展历史集合(不管是否属于 S_1 和 S_2)都是有穷集合, 因此 S_1 和 S_2 也是有穷集合;

(2) 在第 2 步, 依据定义 1 以如下方式检测 S_2 是否 TSO-to-TSO 线性化了 S_1: 对每一对扩展历史 $eh \in S_1$ 和 $eh' \in S_2$, 显然它们之间的一一映射的数目是有穷的, 通过枚举这些映射并检测是否保留了特定动作之间的序, 我们可以判定 eh 是否 TSO-to-TSO 可线性化到 eh'. 通过对 S_1 中的每条扩展历史, 寻找 S_2 中是否有与其满足 TSO-to-TSO 可线性化关系的扩展历史, 我们可以判定 S_2 是否 *k*-限界 TSO-to-TSO 线性化了 S_1. 由于 S_1 和 S_2 都是有穷集合, 因此第 2 步的计算一定终止.

当分别验证并发数据结构以及正则顺序规约之间的 *k*-限界 TSO-to-TSO 可线性化、*k*-限界 TSO-to-SC 可线性化和 *k*-限界 TSO 可线性化时, 需要叙述此时集合 S_1 和 S_2 的定义, 以及如何构造这两个集合. 我们还需要确保对这两个集合的构造过程一定终止.

- 验证 *k*-限界 TSO-to-TSO 可线性化

在验证两个并发数据结构是否符合 *k*-限界 TSO-to-TSO 可线性化时, 集合 S_1 和 S_2 分别是 *k*-*ehistory*($[\![L,n]\!]_{tt}$)和 *k*-*ehistory*($[\![L',n]\!]_{tt}$). 集合 S_1 可以通过如下方法构造.

(1) 首先, 枚举出所有可能的 *k*-扩展历史的集合(不管是否对应并发数据结构的执行). 由于进程数目、函数名集合和数据域都是有穷的, 所以所有可能的 *k*-扩展历史集合是有穷的, 并且可以通过枚举来构造;

(2) 然后, 对每一条 *k*-扩展历史, 使用某种方法检测其是否属于$[\![L,n]\!]_{tt}$ 的某条实际执行. 通过反复使用这种方法判定每一条可能的 *k*-扩展历史, 我们得到了$[\![L,n]\!]_{tt}$ 的实际的 *k*-扩展历史集合.

集合 S_2 可以通过类似的方法处理并发数据结构 L'来构造.

判定一条特定的扩展历史是否属于并发数据结构的实际执行是较为复杂的, 我们将在第 5 节–第 7 节详细叙述判定方法.

- 判定 *k*-限界 TSO-to-SC 可线性化

由定义 2 可以推导出, n 进程下并发数据结构 L'是 *k*-限界 TSO-to-SC 线性化了并发数据结构 L 的, 当且仅当 *k*-*history*($[\![L',n]\!]_{sc}$)是 TSO-to-TSO 线性化了 *k*-*history*($[\![L,n]\!]_{tt}$)的. 这里需要注意的是, 一条历史同时也是一条扩展历史, 因此在两条历史之间也有着 TSO-to-TSO 可线性化关系. 因此, 验证两个并发数据结构是否符合 *k*-限界 TSO-to-SC 可线性化时, 集合 S_1 和 S_2 分别是 *k*-*history*($[\![L,n]\!]_{tt}$)和 *k*-*history*($[\![L',n]\!]_{sc}$).

由于每个调用刷出或返回刷出动作都来自于一个之前发生的函数调用或函数返回动作, 如果扩展历史 *eh*

在函数调用动作和函数返回动作上的投影的长度为 k, 则原扩展历史最长有 $2k$. 因此, 可以通过如下方法计算出 S_1: 先依照之前的方法计算出$[\![L,n]\!]_{tt}$的长度不超过 $2k$ 的扩展历史的集合, 再将这些扩展历史投影到函数调用动作和函数返回动作上, 筛选出长度不超过 k 的历史的集合.

$[\![L',n]\!]_{sc}$ 使用有穷的内存单元集合、有穷的函数名集合、有穷的控制状态集合、限界的进程数目以及有穷的数据域, 并且没有实际使用存储缓冲区. 因此, $[\![L',n]\!]_{sc}$是一个有穷状态的标号迁移系统. 通过将$[\![L',n]\!]_{sc}$中的非函数调用和非函数返回动作都转化为空迁移, 我们可以证明, *history*($[\![L',n]\!]_{sc}$)是一个正则语言. 由于使用同一个函数名集合、进程集合和数据域的可能的 k-历史集合是有穷的, 进而是正则的, 所以可以通过枚举计算出来. 通过计算这两个正则语言的交, 可以得到 S_2=*k-history*($[\![L',n]\!]_{sc}$), 它是一个有穷且正则的集合.

- 判定 k-限界 TSO 可线性化

给定标号迁移系统 A, 令 *k-transHistory*(A)为这样的历史 h 的集合: 存在扩展历史 $eh\in ehistory(A)$, 使得 $|Trans(eh)|\leqslant k$, 且 $h=Trans(eh)$. 由定义 3 可知: n 进程下并发数据结构 L 是 k-限界 TSO 可线性化到正则的顺序规约 *Spec* 的, 当且仅当对 *k-transHistory*($[\![L,n]\!]_{tt}$)中的每条历史 h, 都存在一条顺序历史 $s\in Spec$, h 可以通过先添加零个或若干个函数返回动作, 再去掉不配对的函数调用动作得到历史 h', 使得 h'是 TSO-to-TSO 可线性化到 s 的.

给定历史集合 S, 令 *k-history*(S)表示 S 中 k-历史的集合. 不难证明: n 进程下并发数据结构 L 是 k-限界 TSO 可线性化到顺序规约 *Spec* 的, 当且仅当 *posSet*(*k-transHistory*($[\![L,n]\!]_{tt}$))(一个集合的集合)中至少一个元素是 TSO-to-TSO 可线性化到(k+n)-*history*(*Spec*)的. 这里的 k+n 来自于一个长度为 k 的历史, 最多被添加 n 个函数返回动作. *posSet*(*k-transHistory*($[\![L,n]\!]_{tt}$))集合的每个元素 S 是一个历史集合, 且 S 包含了和 *k-transHistory* ($[\![L,n]\!]_{tt}$)相同数目的历史. S 中的每个历史, 是 *k-transHistory*($[\![L,n]\!]_{tt}$)中的某个历史通过添加零个或若干个函数返回动作, 再去掉不配对的函数调用动作得到的. 而且, *k-transHistory*($[\![L,n]\!]_{tt}$)中的每个历史也能通过添加若干调用, 再去掉若干返回的方式对应到 S 中的某条历史. 因此, 验证并发数据结构 L 是否 k-限界 TSO 可线性化到正则顺序规约 *Spec* 时, 我们进行多次 TSO-to-TSO 可线性化验证, 每次验证使用的集合 S_1 和 S_2 分别是 *posSet*(*k-transHistory*($[\![L,n]\!]_{tt}$))的一个元素和(k+n)-*history*(*Spec*).

每个返回刷出动作来自于之前的一个函数返回动作, 进而关联到之前的一个函数调用动作; 每个返回刷出动作发生前, 对应的调用刷出动作一定已经发生; 一个调用刷出动作来自于之前的一个函数调用动作. 因此, 如果扩展历史 eh 在函数调用动作和返回刷出动作上的投影的长度为 k, 则原扩展历史最长有 $3k$. 可以通过如下方法计算出 *k-transHistory*($[\![L,n]\!]_{tt}$): 先计算出$[\![L,n]\!]_{tt}$的长度不超过 $3k$ 的扩展历史的集合, 然后依次进行 *Tran* 处理, 保留 *Trans* 处理后长度不超过 k 的结果. 之后, 通过枚举 *k-transHistory*($[\![L,n]\!]_{tt}$)中每个历史可能的添加函数返回和删除无配对的函数调用的方式, 我们就得到了 S_1=*posSet*(*k-transHistory*($[\![L,n]\!]_{tt}$)).

因为 *k-transHistory*($[\![L,n]\!]_{tt}$)中只有有穷数目的历史, 且添加的函数返回动作的函数名集合、进程集合和数据域都是有穷的, 所以这个计算过程是终止的. 由于函数名集合、进程集合和数据域都有穷, 可能的(k+n)-历史的集合是有穷的, 进而是正则的, 可以通过枚举计算出来. 因此, 通过计算这个集合和 *Spec* 的交, 可以构造 S_2=*k-history*(*Spec*), 这个集合是有穷的, 也是正则的.

5 修改后的操作语义

根据第 4 节中的思路, 我们需要解决判定 TSO 内存模型下并发数据结构是否有着一条特定的扩展历史的问题. 为此, 在后面章节中参考了文献[8]的思路, 将操作语义和易失通道机器关联起来. 文献[8]中的方法未涉及函数调用、函数返回、调用刷出和返回刷出动作. 处理函数调用和函数返回动作的难点在于, $[\![L,n]\!]_{tt}$ 中的一个函数调用或函数返回动作会同时完成两个任务: 一个任务是将一个函数调用或函数返回标记放入对应进程的存储缓冲区, 一个任务是修改对应进程的控制状态.

本节提出了如何修改原有的客户程序和并发数据结构, 提出了新的内存模型及其操作语义, 它使用“刷

出”动作和 cas 动作来对应扩展历史中的函数调用、函数返回、调用刷出和返回刷出动作. 我们修改后的内存模型和操作语义与$[\![L,n]\!]_{tt}$有着相同的扩展历史集合, 因此, 判断操作语义是否存在一条特定的扩展历史这个问题, 在修改后的操作语义和$[\![L,n]\!]_{tt}$上是等价的问题. 而且由于函数调用、函数返回、调用刷出和返回刷出动作都被转化为刷出和 *cas* 动作, 使得我们更易于使用文献[8]的思路, 将这样的操作语义和易失通道机器关联起来.

5.1 针对调用刷出和返回刷出修改操作语义

令 z_f 是一个新的内存单元. 我们希望将$[\![L,n]\!]_{\mathrm{tt}}$中的一个同时完成两个任务的函数调用动作变成紧邻的两个动作, 其中, 前者是一个只影响对应进程控制状态的“纯粹”的函数调用动作, 后者是一个对 z_f 的写动作. $[\![L,n]\!]_{tt}$中, 函数调用动作的向存储缓冲区中放入函数调用标号的任务, 在新的操作语义中, 实际上由对 z_f的写动作完成. 当 z_f 的写从存储缓冲区中刷出到内存中时, 新的操作语义不是发出刷出动作, 而是发出调用刷出动作. 通过这样的方式, 在新的操作语义中, 我们用对 z_f的写和刷出来对应了$[\![L,n]\!]_{tt}$中的函数调用动作对存储缓冲区的影响.

在新的操作语义中, 我们修改了客户程序和并发数据结构, 添加了对 z_f的写动作. 确切地说, 给定一个并发数据结构 $L=(X_L,M_L,D_L,Q_L,\rightarrow_L)$, 新的操作语义是一个标号迁移系统$[\![C_f(Mod(L)),n]\!]_f$. 这里的 C_f是一个映射函数, 它将每个进程 P_i映射到一个修改后的最一般客户 $Mod(MGC)$. $[\![C_f(Mod(L)),n]\!]_f$中改进的最一般客户使用修改后的并发数据结构 $Mod(L)$, 其中,

- 修改后的最一般客户 $Mode(MGC)$是一个五元组$(\{z_f\},M,D_L\cup\{call,ret\},\{in_{clt},inl_{ib1},in_{lib2}\},\rightarrow)$.

我们扩展了原有的最一般客户, 在函数返回动作发生后加入了一个对 z_f 的写动作, 写入一个函数返回标记. 迁移关系$\rightarrow$被定义为$\rightarrow=\{(in_{clt},call(m,a),in_{lib1}),(in_{lib1},return(m,a),in_{lib2}),(in_{lib2},write(z_f,ret),in_{clt})|m\in M,a\in D_L\}$;

- 修改后的并发数据结构 $Mod(L)$是一个五元组$(X_L\cup\{z_f\},M,D_L\cup\{call,ret\},Q_L\cup Q',\rightarrow'_L)$.

修改后的并发数据结构在每个函数调用动作之后, 先执行一个对 z_f 的写动作, 写入函数调用标记. 确切地说, 迁移关系 $\rightarrow'_L$ 是通过对迁移关系$\rightarrow_L$ 进行如下修改而得到的: 任取 $a\in D_L$, $m\in M$ 且 $q\in Q_L$, 如果有着 $is_{(m,a)}\xrightarrow{act}_L q_1$ 迁移关系, 则将其修改为 $is_{(m,a)}\xrightarrow{write(z_f,call)}'_L q_{(m,a,q_1)}$ 和 $q_{(m,a,q_1)}\xrightarrow{act}'_L q_1$. 状态集合 Q'就是这些新加入的中间状态 $q_{(m,a,q_1)}$ 的集合. 给定并发数据结构 L 的原操作语义$[\![C(L),n]\!]_{tt}=(Conf_{tt},\Sigma_{tt},\rightarrow_{tt},InitConf_{tt})$, 修改后的操作语义$[\![C_f(Mod(L)),n]\!]_f$仍是一个标号迁移系统$(Conf_f,\Sigma_f,\rightarrow_f,InitConf_f)$. 这里的下标 f 表明, 这个操作语义是为了解决调用刷出和返回刷出动作而提出的. 状态集合 $Conf_f$是通过向状态集合 $Conf_{tt}$中加入内存单元 z_f并修改最一般客户和并发数据结构的控制状态(如上面所述)得到的, 迁移标号集合Σ_f 是通过向迁移标号集合Σ_{tt}中加入内存单元 z_f的写得到的, 格局 $InitConf_f$是通过向格局 $InitConf_{tt}$中加入内存单元 z_f的求值得到的.

迁移关系$\rightarrow_f$是通过对迁移关系$\rightarrow_{tt}$进行如下修改得到的.

- 在进行函数调用和函数返回迁移时, 只修改当前进程的控制状态, 而不将函数调用标记或函数返回标记放入进程的存储缓冲区. 以函数调用迁移为例, $[\![C_f(Mod(L)),n]\!]_f$ 中, 对函数调用和函数返回的迁移规则如下, 我们可以看到, 只修改了进程的控制状态:

$$\frac{p(i)=in_{clt}}{(p,d,u)\xrightarrow{call(i,m,a)}_t(p[i:(is_{(m,a)},in_{lib1})],d,u)},\quad \frac{p(i)=(fs_{(m,a)},in_{lib1})}{(p,d,u)\xrightarrow{return(i,m,a)}_t(p[i:in_{lib2}],d,u)};$$

- 修改$[\![L,n]\!]_{tt}$中调用刷出、返回刷出和刷出迁移的迁移规则. 当存储缓冲区中 z_f的写刷出时, 如果对应的数据是 *call*, 则发动调用刷出迁移; 如果对应的数据是 *ret*, 则发动返回刷出迁移. 具体的迁移规则如下:

$$\frac{u(i)=l\cdot(z_f,call)}{(p,d,u)\xrightarrow{flushCall(i)}_f(p,d[z_f:call],u[i:l])},\quad \frac{u(i)=l\cdot(z_f,ret)}{(p,d,u)\xrightarrow{flush\,Return(i)}_f(p,d[z_f:ret],u[i:l])};$$

- 当存储缓冲区中其他内存单元的写刷出时, 发动刷出迁移:

$$\frac{u(i)=l\cdot(x,a),x\neq z_f}{(p,d,u)\xrightarrow{flush(i,x,a)}_f(p,d[x:a],u[i:l])}.$$

下面的引理说明了操作语义$[\![C_f(Mod(L)),n]\!]_f$和操作语义$[\![L,n]\!]_{tt}$有着相同的扩展历史集合. 这个引理的证明可以在本文的技术报告版本[29]中找到.

引理 1. $ehistory([\![C_f(Mod(L)),n]\!]_f)=ehistory([\![L,n]\!]_{tt})$.

5.2 针对函数调用和函数返回修改内存模型和操作语义

在原本的 TSO 内存模型下, 使用读、写或 *cas* 动作对应函数调用和函数返回动作有着固有的困难, 这个困难来自于一个进程无法“及时观察”到其他进程做了函数调用或函数返回动作. 对此的解释如下: 由于一个进程 P 的函数调用和函数返回动作不影响存储缓冲区和内存, 因此其他进程是观测不到进程 P 的函数调用或函数返回动作的. 如果通过在进程 P 的函数调用和函数返回动作之后加入对某个内存单元 x 的写动作来产生其他进程可见的动作来完成这个目标, 则这个写动作需要在进程 P 的存储缓冲区中的其他数据项都被刷出后才能被刷出. 假定我们希望复现这样的执行, 某个进程 P 在执行函数调用之前, 其存储缓冲区已经有一个某内存单元 y 的写, 这个进程在之后的执行中从来没有刷出这个对 y 的写, 而且正因为这些 y 的写一直没刷出, 另一个进程 P'的函数才能两次返回特定值(假定这两次函数返回之间执行了 *cas* 指令). 当我们尝试复现历史时, 会发现复现的历史“不真实”, 因为在复现的历史中进程 P 对 x 的写一定在进程 P'的第 1 个函数返回之后才可能刷出到内存, 这导致在复现的历史中进程 P 的这个函数调用的位置在进程 P'的第 1 个函数返回之后. 如果通过在进程 P 的函数调用和函数返回动作之后加入 *cas* 动作来完成这个目标, 由于 *cas* 动作会清空存储缓冲区, 所以同样无法复现上面提到的执行. 为了解决这个问题, 文献[18]修改了内存模型本身. 在修改后的内存模型中, 我们将函数调用和函数返回动作与特定的 *cas* 动作“绑定”在一起. 确切地说, 引入了一个“观察者”进程和一个新的内存单元 z_w, 观察者进程不断地用 *cas* 动作修改 z_w, 修改的值是非确定猜测的; 内存模型要求在观察者进程每次对 z_w 的 *cas* 动作后, 下一个动作必须是对应的函数调用或函数返回动作.只有观察者进程修改 z_w 的 *cas* 动作必须被对应的进程的函数调用或函数返回动作响应, 而其他内存单元的刷出和 *cas* 动作不会让其他进程进行响应. 本节叙述如何通过这个思想来修改第 5.1 节构造的操作语义.

令 $markedVal(M,D_L,n)=\{call(i,m,a),return(i,m,a)|1\leqslant i\leqslant n,a\in D_L,m\in M\}$为 z_w 做 *cas* 动作使用的值, 为了对应函数调用和函数返回动作, 这些值本身就记录了函数调用或函数返回. 给定并发数据结构 $L=(X_L,M_L,D_L,Q_L,\rightarrow_L)$, 我们修改并发系统如下.

- 进程 P_1 到 P_n 仍然运行修改后的最一般客户;
- 观察者进程 P_{n+1} 运行如下客户程序: $Cmarked=(\{z_w\},M,markedVal(M,D_L,n),\{q_{wit}\},\rightarrow_{wit})$, 这里, 迁移关系$\rightarrow_{wit}$ 定义为$\{q_{wit},cas_suc(z_w,_,a),q_{wit}|a\in markedVal(M,D_L,n)\}$. 在观察者进程中, 我们只考虑成功的 *cas* 动作, 可以认为另有一个陷阱状态, 当 *cas* 动作失败时, 观察者进程进入陷阱状态并终止执行.

令$[\![C_f(Mod(L)),n]\!]_f=(Conf_f,\Sigma_f,\rightarrow_f,InitConf_f)$为第 5.1 节修改后的操作语义. 本节提出的修改后的操作语义$[\![Clt_f(Mod(L)),n+1]\!]_b$ 是一个标号迁移系统$(Conf_b,\Sigma_b,\rightarrow_b,InitConf_b)$. 这里的 Clt_f 是一个映射函数, 它将进程 P_1 到 P_n 映射到修改后的最一般客户, 将进程 P_{n+1} 映射到 C_{marked}. $Conf_b$ 中的一个格局是一个四元组(p,d,u,mak), 其中, 三元组(p,d,u)是通过向 $Conf_f$ 中的某个格局添加内存单元 z_w 和观察者进程的控制状态得到的, 而 *mak* 是一个取自$\{\perp\}\cup markedVal(M,D_L,n)$集合中的元素, 它用来确保观察者进程的每个 *cas* 动作都被绑定到对应的函数调用或函数返回动作. $\perp$是一个特定的值. 字母表Σ_b 是通过向字母表Σ_f中加入 z_w 的动作而得到的. 初始格局 $InitConf_b$ 是通过向格局 $InitConf_f$ 中加入 z_w 的求值和观察者进程的控制状态而得到的. 迁移关系$\rightarrow_b$ 是通过以如下方式改动迁移关系$\rightarrow_f$而得到的.

- 进程 P_{n+1}, 即观察者进程, 对应的迁移规则如下:

$$\frac{p(n+1)=q_{wit},q_{wit}\xrightarrow{cas\_suc(z_w,a,b)}_{wit}q_{wit},a,b\in markedVal(\mathcal{M},\mathcal{D}_{\mathcal{L}},n),d(z_w)=a,u(n+1)=\varepsilon}{(p,d,u,\perp)\xrightarrow{cas(n+1,z_w,a,b)}_b(p,d[z_w:b],u,b)}.$$

这里, $\rightarrow_{wit}$ 是观察者进程的客户程序 C_{marked} 的迁移关系.

注意, 只有这种迁移会将当前格局的第 4 个元组从⊥修改为 $markedVal(M,D_L,n)$中的值;

- 除函数调用、函数返回动作之外, 进程 P_1 到 P_n 其他的迁移规则不变, 且这些迁移只能在当前格局的第 4 个元组为⊥时发生;
- 一个函数调用迁移 $call(i,m,a)$只有当格局的第 4 个元组的值为 $call(i,m,a)$时才能发生, 并且这个函数调用迁移将会把这个元组的值置为⊥. 函数调用动作的迁移规则如下:

$$\frac{p(i)=in_{clt}}{(p,d,u,call(i,m,a))\xrightarrow{call(i,m,a)}_b(p[i:(is_{(m,a)},in_{lib1})],d,u,\perp)}.$$

函数返回动作的迁移规则类似.

通过对当前格局的第 4 个元组的要求, 我们将观察者进程的对 z_w 的 *cas* 动作和进程 P_1 到 P_n 的函数调用和函数返回动作绑定在了一起. 下面引理说明$[\![Clt_f(Mod(L)),n+1]\!]_b$ 和$[\![C_f(Mod(L)),n]\!]_f$ 有着相同的扩展历史集合. 这个引理的证明可以在本文的技术报告版本[29]中找到.

引理 2. $ehistory([\![Clt_f(Mod(L)),n+1]\!]_b)=ehistory([\![C_f(Mod(L)),n]\!]_f)$.

6 构造对应特定扩展历史的通道机器

本节引入(易失)通道机器的定义. 之后, 我们先用一个例子直观说明如何使用通道机器模拟$[\![Clt_f(Mod(L)),n+1]\!]_b$ 中对应特定扩展历史的执行中特定进程的行为, 然后给出这样的通道机器的构造.

6.1 (易失)通道机器的定义

一个经典的通道机器包含了有穷的控制状态和一个容量无界的通道, 通道机器的一步迁移在修改控制状态的同时, 也可以向通道中放入数据或者从通道中取出数据. 一个易失通道是一个在任意时刻都可能有任意数量的数据从通道中丢失的通道, 且这个数据丢失过程不会对通道机器有任何通知. 一个易失通道机器是一个使用易失通道的通道机器. 文献[8]在如下几个方面扩展了(易失)通道机器的定义.

- 每一个迁移中加入了正则卫士表达式, 判断某个通道的内容是否属于一个正则语言;
- 在每一个迁移发生之前, 允许将某个通道的元素进行替换;
- 引入了强符号的概念. 强符号是通道内容的一部分, 它们在迁移过程中是不会丢失的, 并且一个通道内的强符号数目有着上界.

在文献[18]中, 我们稍微扩展了文献[8]中(易失)通道机器的定义, 允许使用多种强符号, 并对每个通道里的每种强符号的数目分别限界. 本文使用文献[18]中的(易失)通道机器定义.

一个正则卫士表达式形如 $c\in L$, 这里, c 是一个通道名, L 是一个正则集合. 给定一个序列 l, 如果 $u\in L$, 则称 $u\vDash c\in L$ 成立. 为了表示的简便, 我们把正则卫士表达式 $c\in\Sigma^*a\Sigma^*$简写为 $a\in c$, 把正则卫士表达式 $c\in\varepsilon$简写为 $c=\varepsilon$, 把正则卫士表达式 $c\in\Sigma^*$简写为 $c:\Sigma$. 一个通道集合 C 上的正则卫士表达式为每个通道 $c\in C$ 关联一个正则卫士表达式. 我们使用 $Guard(C)$表示通道集合 C 上的正则卫士表达式的集合. 关系⊨也被扩展到 $Guard(C)$上. 给定函数 u 为每个通道分配内容以及 $g\in Guard(C)$, 如果对每个通道 $c\in C$, $u(c)\vDash g(c)$都成立, 则称 $u\vDash g$ 成立.

通道 c 上有 3 种操作: *nop* 操作不修改通道, $c[\sigma]!a$ 操作先依据替换σ进行元素替换再向通道中插入数据 a, $c?a$ 操作从通道中取出数据 a. 这里, σ是一个元素替换, 当σ把每个元素替换为自己时, 将此时的 $c[\sigma]!a$ 操作简写为 $c!a$. 我们为每一种通道操作定义一个序列之间的关系如下: 任取有穷序列 u 和 u', 如果 $u=u'$, 则$[\![nop]\!](u,u')$成立; 如果 $u'=a\cdot u[\sigma]$, 则$[\![c[\sigma]!a]\!](u,u')$成立; 如果 $u=u'\cdot a$, 则$[\![c?a]\!](u,u')$成立. 一个通道集合 C 上的通道操作为每个通道 $c\in C$ 关联一个通道操作. 我们使用 $Op(C)$表示通道集合 C 上的通道操作的集合. 关系$[\![\cdot]\!]$也被扩展到 $Op(C)$上. 给定为每个通道分配内容的函数 u 和 u'以及通道集合 C 上的通道操作 $op\in Op(C)$, 如果对每个通道, $[\![op(c)]\!](u(c),u'(c))$都成立, 则$[\![op]\!](u,u')$成立.

一个通道机器 M 被定义为一个五元组 $M=(Q,CH,\Sigma_{CH},\Lambda,\Delta)$, 其中, Q 是一个有穷的状态集合, CH 是一个有

穷的通道名集合，Σ_{CH} 是通道内容的有穷字母表，Λ是有穷的迁移标号集合，$\Delta\subseteq Q\times(\Lambda\cup\{\varepsilon\})\times Guard(CH)\times Op(CH)\times Q$ 是有穷的迁移关系集合. 为了阅读的方便，我们把$(q,l,g,op,q')\in\Delta$写为 $q\xrightarrow{l,g,op}q'$. 通道机器 M 的操作语义是一个标号迁移系统$(Conf,\Lambda\cup\{\varepsilon\},\rightarrow_M,initConf)$. 每个格局是一个二元组$(q,u)$，其中，$q\in Q$ 且 $u:CH\rightarrow\Sigma_{CH}^*$. 如果存在通道集合 CH 上的通道操作 op，使得 $q\xrightarrow{l,g,op}q'$，$u\vDash g$ 且$[\![op]\!](u,u')$都成立，则迁移$((q,u),l,(q',u'))$属于迁移关系$\rightarrow_M$.

给定有穷序列 l 和 l'，如果可以通过删除 l'的零个或若干个元素得到 l，则称 l 是 l'的子串. 令 $S=\langle SET_1,\ldots,SET_m\rangle$为一个向量，其中每个 SET_i 都是一个有穷集合，代表一种强符号，且 m 是一个正整数. 令 $K=\langle k_1,\ldots,k_m\rangle$为一个向量，其中每个元素都是一个自然数，代表对应种类的强符号在一个通道中的数目限制. 下面定义带有强符号约束(S,K)的易失通道机器，称为(S,K)-易失通道机器. 给定 $u,u':CH\rightarrow\Sigma_{CH}^*$，如果对任意通道 $c\in CH$ 和 $1\leqslant i\leqslant m$，$u(c)$是 $u'(c)$的子串，$u(c)\uparrow_{S(i)}=u'(c)\uparrow_{S(i)}$，且 $u(c)$在 $S(i)$上的投影的长度小于等于 $K(i)$，则称 u 和 u'满足关系 $u\preccurlyeq_S^K u'$. (S,K)-易失通道机器是使用易失通道并遵守强符号约束(S,K)的通道机器，它的操作语义是一个标号迁移系统$(Conf,\Lambda\cup\{\varepsilon\},\rightarrow'_M,initConf)$. 给定 $q,q'\in Q$，$u,u':CH\rightarrow\Sigma_{CH}^*$ 和 $l\in\Lambda\cup\{\varepsilon\}$，如果存在 $v,v':CH\rightarrow\Sigma_{CH}^*$，使得 $v\preccurlyeq_S^K u$，$((q,v),l,(q',v'))\in\rightarrow'_M$ 和 $u'\preccurlyeq_S^K v'$ 都成立，则迁移$((q,u),l,(q',u'))$属于迁移关系 $\rightarrow'_M$.

给定通道机器 M 的状态 q 和 q'，如果将 M 视为(S,K)-易失通道机器，我们称其在操作语义上从格局(q,c_{init})到格局(q',c_{init})的迹的集合为 $LT_{q,q'}^{S,K}(M)$. 这里，c_{init} 的作用是将每个通道名映射到ε. (S,K)-易失通道机器的控制状态可达问题是指：给定通道机器 M、强符号限制(S,K)和状态 q 和 q'，判定 $LT_{q,q'}^{S,K}(M)$ 是否为空集. 文献[8]说明了控制状态可达问题是可判定的，其考虑的易失通道机器相当于 S 只包含一个集合时的情况. 使用类似的思想不难证明，(S,K)-易失通道机器的控制状态可达问题是可判定的. 将 M 视为(S,K)-易失通道机器时，我们称其在操作语义上从格局(q,c_{init})到格局(q',c_{init})，且通道不丢失任何内容的迹的集合为 $T_{q,q'}^{S,K}(M)$. 或者说，$T_{q,q'}^{S,K}(M)$ 是“非易失通道”版本的 $LT_{q,q'}^{S,K}(M)$.

给定两个采用不同通道的通道机器 $M=(Q,CH,\Sigma_{CH},\Lambda,\Delta)$和 $M'=(Q',CH',\Sigma_{CH},\Lambda,\Delta')$，即 $CH\cap CH'=\varnothing$，M 和 M'的乘积被定义为另一个通道机器 $M\otimes M'=(Q\times Q',CH\cup CH',\Sigma_{CH},\Lambda,\Delta'')$. Δ''中的一步非ε迁移会同时完成 M 和 M'中使用同样标号的两个迁移，此时，新的卫士表达式是这两个卫士表达式的交，新的通道操作正好包含这两个迁移的通道操作；Δ''中的一步ε迁移完成 M 或 M'中的一步ε迁移. 文献[8]中给出了如下引理，说明了两个易失通道机器乘积的 LT 集合等于两个易失通道机器各自的 LT 集合的交：

引理 3. 给定 q_1,q_1',q_2,q_2' 以及 S 和 K，$LT_{q,q'}^{S,K}(M_1\otimes M_2)=LT_{q_1,q_1'}^{S,K}(M_1)\cap LT_{q_1,q_1'}^{S,K}(M_1)$.

这里，$q=(q_1,q_2)$且 $q'=(q_1',q_2')$.

6.2 使用通道机器模拟$[\![Cltf(Mod(L)),n+1]\!]_b$执行一条特定的扩展历史

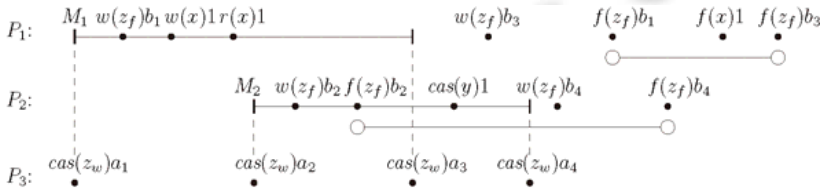
本节通过一个例子说明如何使用通道机器模拟并发系统对应一条特定的扩展历史的执行. 给定并发系统$[\![Clt_f(Mod(L)),n+1]\!]_b$的一条 k-扩展历史 eh，我们为每个进程 P_i 构造一个通道机器 M_i^{eh} 来模拟进程 P_i 的行为. 下一节将证明易失通道机器 $M_1^{eh-w}\otimes\ldots\otimes M_{n+1}^{eh-w}$ 模拟了并发系统对应 eh 的执行. 这里，M_i^{eh-w} 是从 M_i^{eh} 构造得到的通道机器. M_i^{eh} 的定义见第 6.3 节，M_i^{eh-w} 的定义见第 7 节.

通道机器 M_i^{eh} 需要包含进程 P_i 的状态，包括客户程序状态和并发数据结构状态，并依据这些状态进行迁移. 为了进行通道机器的同步，M_i^{eh} 也需要处理其他进程的动作，为此，它不断非确定地猜测其他进程的动作. 其他进程的写动作和 cas 动作都会被猜测为一个对应的写动作. 为了保证 M_i^{eh} 对应的扩展历史一定是 eh(或其前缀)，我们向 M_i^{eh} 的状态加入了一个元组来确保这一点. M_i^{eh} 只有一个通道，对应了进程 P_i 的存储缓冲区. M_i^{eh} 在猜测其他进程的写动作时，也会向通道中放入对应的内存单元和值. 为了保证丢失部分通道内容不影响执行，通道中的每个元素都包含内存的快照.

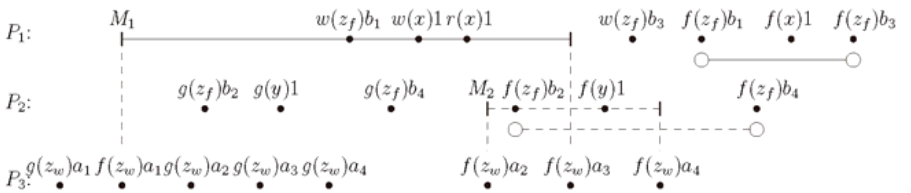
M_i^{eh} 会将进程 P_{n+1} 对 z_w 的 cas 动作猜测为对应的写动作并向通道写入 z_w，这些写入的 z_w 被设置为强符号.

它们离开通道时将触发刷出迁移，并且 M_i^{eh} 在下一步的迁移一定是对应的函数调用或函数返回动作(不管这些函数调用和函数返回动作是否属于进程 P_i). M_i^{eh} 在函数调用和函数返回动作后会向通道写入 z_f，也会猜测其他进程对 z_f 的写动作. 这些写入的 z_f 也被设置为强符号，它们离开通道将触发调用刷出或返回刷出迁移. 这就是 M_i^{eh} 捕捉扩展历史的机制.

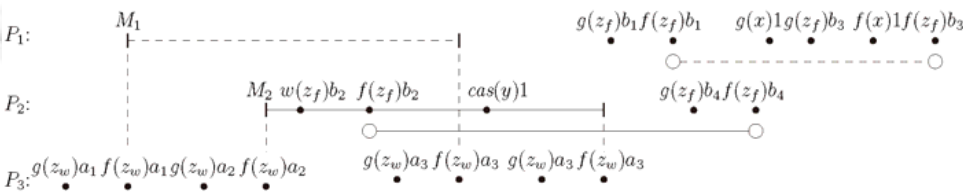
图1给出了一个 M_i^{eh} 模拟并发系统中进程 P_i 执行的例子. 这里, *eh* 是一个包含 8 个动作的扩展历史，图1(a)给出了 *eh* 对应的并发系统 $[\![Clt_f(Mod(L)),3]\!]_b$ 的执行，而图 1(b)~图 1(d)分别给出了 M_1^{eh}, M_2^{eh} 和 M_3^{eh} 的对应执行，其中，进程 P_3 是观察者进程.



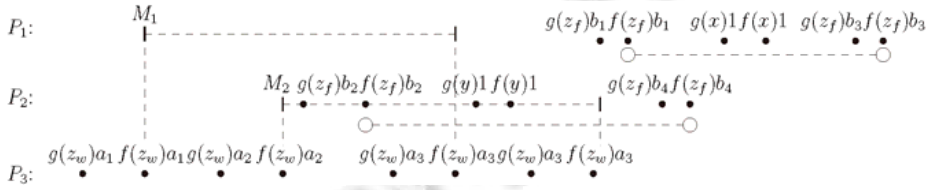
(a) $[\![Clt_f(Mod(L)),3]\!]_f$ 的一条执行 t



(b) 对应 t 的通道机器 M_1^{eh} 的执行 t_1



(c) 对应 t 的通道机器 M_2^{eh} 的执行 t_2



(d) 对应 t 的通道机器 M_3^{eh} 的执行 t_3

图 1 $[\![Clt_f(Mod(L)),3]\!]_b$ 的一条执行以及通道机器 M_1^{eh}, M_2^{eh} 和 M_3^{eh} 对应这条执行的执行

图1中的每条执行按照时间顺序从左到右画出动作，依据属于不同的进程从上到下画动作. 为了强调每一对函数调用和函数返回动作，我们在每个函数调用和函数返回动作下面画一条竖线，且用横线连接一对函数调用和函数返回动作对应的竖线. 为了强调每一对调用刷出和返回刷出动作，在每个调用刷出和返回刷出动作下面画一个圆圈，且用横线连接一对调用刷出和返回刷出对应的圆圈. 当函数调用、函数返回、调用刷出和返回刷出动作是进程 P_i 的动作时，对应的横线画为实线；当函数调用、函数返回、调用刷出和返回刷出动作来自猜测其他进程的动作时，画为虚线.

图1中的术语解释如下: 为了节省空间，我们用 $w(x)1$ 表示将 x 的值写为 1 的动作，用 $r(x)1$ 表示从 x 读到

1 的动作, 用 $f(x)1$ 表示从存储缓冲区中刷出一项且其将 x 写为 1 的动作, 用 $cas(y)1$ 表示使用 cas 成功地将 y 的值修改为 1 的动作. 通道机器 M_i^{eh} 用 $g(z_f)b_2$ 和 $g(z_w)a_1$ 分别表示两个猜测的写动作: 前者将 z_f 的值写为 b_2, 后者将 z_w 的值写为 a_1. 可以看到: 一些猜测的写(如 $g(z_f)b_2$)在并发系统的执行中对应写动作(如 $w(z_f)b_2$), 另一些猜测的写(如 $g(z_w)a_1$)在并发系统的执行中对应 cas 动作(如 $cas(z_w)a_1$). 为了节省空间, 图上用 a_1~a_4 分别代表向 z_w 写入的 4 个函数调用或函数返回的值, 用 b_1~b_4 分别代表向 z_f 写下的 4 个标号, 即 $a_1=call(1,M_1,\_)$, $a_2=call(2,M_2,\_)$, $a_3=return(1,M_1,\_)$, $a_4=return(2,M_2,\_)$, $b_1=b_2=call$ 且 $b_3=b_4=ret$. 需要注意的是: 在图 1(b)~图 1(d)中的动作来自 M_i^{eh}, 因此每个动作还带有内存快照. 为了理解方便, 我们仍沿用图 1(a)的写法.

我们分别称图 1(a)~图 1(d)中的执行为 t, t_1~t_3. 可以看到, 各个进程模拟的执行(即 t_1~t_3)和原执行 t 有着相同的对内存单元修改动作的序列. 这里的内存单元修改指刷出和 cas 动作, 这里不对两者进行区分. 然而, t_1 和 t 有一点很大的不同在于: t 中有多个存储缓冲区, 而 t_1 只有一个通道. 为了保证相同的内存修改动作的序列, t_1 中猜测其他进程动作的写时, 可能会让一些写“提前”发生. 例如: 在 t 中, P_1 中的 $w(x)1$ 先发生但 P_2 中的 $cas(y)1$ 先修改内存; 而在 t_1 中, 只能先让猜测的对 y 的写先发生, 这样对 y 的修改才能先发生.

6.3 通道机器 M_i^{eh}

本节简述通道机器 M_i^{eh} 的构造. 在本文的技术报告版本[29]中, 可以找到 M_i^{eh} 的详细定义.

给定一条长度为 k 的扩展历史 $eh=\alpha_1\cdot\ldots\cdot\alpha_k$, 可以构造一个识别 eh 的确定型有穷自动机 $A_{eh}=(Q_s,\Sigma_s,\rightarrow_s,F_s,q_{s_0})$. 这里, $Q=\{q_{s_0},...,q_{s_k}\}$ 是有穷的状态集合, $\Sigma_s=\{call(i,m,a),return(i,m,a),flushCall(i),flushReturn(i)|1\leqslant i\leqslant n, a\in D,m\in M\}$是迁移标号的集合且 M 和 D 是对应的函数名集合和数据域, q_{s_0} 是初始状态, $F_s=\{q_{s_k}\}$ 是接收状态的集合, 迁移关系$\rightarrow_s$包含迁移 $q_{s_0}\xrightarrow{\alpha_1}{}_s\, q_{s_1},...,q_{s_{k-1}}\xrightarrow{\alpha_k}{}_s\, q_{s_k}$.

给定并发数据结构 $L=(X_L,M_L,D_L,Q_L,\rightarrow_L)$, 令修改后的并发数据结构 $Mod(L)$为五元组$(X_L\cup\{z_f\},M,D_L\cup\{call,ret\},Q_L\cup Q',\rightarrow'_M)$. 令 Val 是一个为内存单元分配值的函数的集合, 且每个属于 Val 的函数将 X_L 中的内存单元映射到 D_L 中元素, 将 z_w 映射到$\{\perp\}\cup markedVal(M,D_L,n)$中的元素, 将 z_f 映射到$\{\perp,call,ret\}$中的元素. 对 $i\neq n+1$, 通道机器 M_i^{eh} 定义为五元组 $(Q_i^{eh},\{c_i\},\Sigma,\Lambda,\Delta_i^{eh})$. M_i^{eh} 只使用一个通道 c_i, 而 Q_i^{eh}, Σ和Λ定义如下.

- $Q_i^{eh}=(\{in_{clt},in_{lib2}\}\cup((Q_L\cup Q')\times\{in_{lib1}\})\times Val\times Val\times Q_s\times\{\perp\}\cup markedVal(M,D_L,n))$是状态的集合, 这里, in_{clt}, in_{lib1} 和 in_{lib2} 是修改后的最一般客户的状态. 一个 Q_i^{eh} 中的状态是一个五元组(q,d_c,d_g,q_s,mak), 其中: q 是修改后的最一般客户和修改后的并发数据结构的控制状态; d_c 存储一个内存单元求值函数, 表示并发系统当前的内存求值; d_g 存储另一个内存单元求值函数, 它是通过使用将通道 c_i 缓存的所有写更新 d_c 得到的; q_s 是有穷自动机 A_{eh} 的状态, 用来记录当前执行产生的扩展历史; mak 是一个标记, 用来确保 M_i^{eh} 每次从通道中取出一个 z_w 的写(来自 M_i^{eh} 猜测进程 P_{n+1} 对 z_w 的 cas 动作)之后都紧跟着一个对应的函数调用或函数返回动作;
- 通道字母表 $\Sigma=\Sigma_{s_1}\cup\Sigma_{s_2}\cup\Sigma_{s_3}$ 是 3 个集合的并集.其中 $\Sigma_{s_1}=\{(n+1,z_w,d)|d\in Val\}\{(j,z_f,d)|1\leqslant j\leqslant n,d\in Val\}$ 中的一个元素代表一个猜测的对 z_w 的写(对应进程 P_{n+1} 上对 z_w 的 cas 操作), 或者是进程 P_i 对 z_f 的写(在函数调用和函数返回之后), 或者是猜测的其他进程对 z_f 的写. $\Sigma_{s_2}=\{((j,x,d),\#)|1\leqslant j\leqslant n,x\in X_L,d\in Val\}$中的一个元素代表当前缓存的进程 P_i 对相应内存单元的最新的写. 而 $\Sigma_{s_3}=\{a|(a,\#)\in\Sigma_{s_2}\}$ 中的一个元素代表当前缓存的对相应内存的一个写, 且这个写要么是进程 P_i 猜测的(其他进程的写), 要么这个写是进程 P_i 执行的写, 但不是进程 P_i 对这个内存单元的最新写;
- Λ是迁移标号的集合, 包含写、cas、刷出、函数调用、函数返回、调用刷出、返回刷出和ε迁移. τ动作和读动作对应的迁移的标号都是ε.

$\Delta_i^{eh}\in Q_i^{eh}\times(\Lambda\cup\{\varepsilon\}\times Guard(\{c_i\})\times Op(\{c_i\}))\times Q_i^{eh}$ 是迁移关系. 在本文的技术报告版本[29]中, 可以找到 Δ_i^{eh} 的详细定义. 由于空间所限, 本文只展示如下迁移的构造: 并发数据结构和客户程序对 z_f 的写、猜测其他进程对 z_f 的写、调用刷出(来自从通道中取出 z_f 的写)、猜测进程 P_{n+1} 对 z_w 的写、刷出对 z_w 的写、函数调用以及

猜测函数调用. 任取 $q\in\{in_{clt},in_{lib2}\}\cup((Q_L\cup Q')\times\{in_{lib1}\})$, $q_t\in Q'$, $d_c,d_g\in Val$ 和 $q_s\in Q_s$:

- 并发数据结构对 z_f 的写: 令 $is_{(m,a)}\xrightarrow{write(z_f,call)}_{\mathcal{L}}{}' q_t$ 为 $Mod(L)$的迁移, 则:

$$((is_{(m,a)},in_{lib1}),d_c,d_g,q_s,\perp)\xrightarrow{op,c_i:\Sigma,c_i!\beta}_{A_i^{eh}}((q_t,in_{lib1}),d_c,d'_g,q_s,\perp).$$

  这里, $d'_g=d_g[z_f:call]$, $\beta=(i,z_f,d'_g)$, 迁移标号 $op=write(i,z_f,d'_g)$. 注意, 放入通道的每个数据项包含写动作发生的进程号、被修改的内存单元以及一个内存快照. 被修改的内存单元的新值可以从内存快照中读取;

- 客户程序对 z_f 的写:

$$(in_{lib2},d_c,d_g,q_s,\perp)\xrightarrow{op,c_i:\Sigma,c_i!\beta}_{A_i^{eh}}(in_{clt},d_c,d'_g,q_s,\perp).$$

  这里, $d'_g=d_g[z_f:ret]$, $\beta=(i,z_f,d'_g)$, 迁移标号 $op=write(i,z_f,d'_g)$;

- M_i^{eh} 猜测其他进程的一个对 z_f 的写: 令 $1\leqslant j\leqslant n$, $j\neq i$ 且 $a\in\{call,ret\}$, 则:

$$(q,d_c,d_g,q_s,\perp)\xrightarrow{op,c_1:\Sigma,c_1!\beta}_{A_i^{eh}}(q,d_c,d'_g,q_s,\perp).$$

  这里, $d'_g=d_g[z_f:a]$, $\beta=(j,z_f,d'_g)$, 迁移标号 $op=write(j,z_f,d'_g)$. 这个猜测不影响 M_i^{eh} 中修改后的并发数据结构或客户程序的控制状态;

- 刷出一个对 z_f 赋值为 $call$ 的写, 引发一个调用刷出迁移: 令 $1\leqslant j\leqslant n$, 如果 $q_s\xrightarrow{flushCall(j)}_s q'_s$, 则:

$$(q,d_c,d_g,q_s,\perp)\xrightarrow{op,c_i:\Sigma,c_i?(j,z_f,d)}_{A_i^{eh}}(q,d,d_g,q'_s,\perp).$$

  这里, $d(z_f)=call$, 迁移标号 $op=flushCall(j)$;

- M_i^{eh} 猜测进程 P_{n+1} 的一个对 z_w 的写: 令 $a\in markedVal(M,D_L,n)$, 则:

$$(q,d_c,d_g,q_s,\perp)\xrightarrow{op,c_i:\Sigma,c_i!\beta}_{A_i^{eh}}(q,d_c,d'_g,q_s,\perp).$$

  这里, $d'_g=d_g[z_w:a]$, $\beta=(n+1,z_w,d'_g)$, 且迁移标号 $op=write(n+1,z_w,d'_g)$. 这个猜测不影响 M_i^{eh} 中修改后的并发数据结构或客户程序的控制状态;

- 刷出通道中缓存的对 z_w 的一个写: 以这个写将 z_w 赋值为一个函数调用为例:

$$(q,d_c,d_g,q_s,\perp)\xrightarrow{op,c_i:\Sigma,c_i?(n+1,z_w,d)}_{A_i^{eh}}(q,d,d_g,q_s,call(j,m,c)).$$

  这里, $d(z_w)=call(j,m,c)$, 且迁移标号 $op=flush(n+1,z_w,d)$. 这个迁移修改了 M_i^{eh} 的状态的第 5 个元组, 将其设置为一个函数调用的信息;

- 进程 P_i 做函数调用: 令 $q_s\xrightarrow{call(i,m,a)}_s q'_s$, 则:

$$(in_{clt},d_c,d_g,q_s,call(i,m,a))\xrightarrow{call(i,m,a),c_i:\Sigma,nop}_{A_i^{eh}}((is_{(m,a)},in_{lib1}),d_c,d_g,q'_s,\perp).$$

  这里, nop 不改变通道 c_i 的内容. 这个迁移将 M_i^{eh} 的状态的第 5 个元组赋值为$\perp$;

- 猜测其他进程的函数调用: 令 $q_s\xrightarrow{call(j,m,a)}_s q'_s$, $1\leqslant j\leqslant n$ 且 $j\neq i$, 则:

$$(q,d_c,d_g,q_s,call(j,m,a))\xrightarrow{call(j,m,a),c_i:\Sigma,nop}_{A_i^{eh}}(q,d_c,d_g,q'_s,\perp).$$

  这个迁移将 M_i^{eh} 的状态的第 5 个元组赋值为$\perp$.

7 TSO 内存模型下可线性化的 3 种限界版本的可判定性和复杂度

如果把$[\![Clt_f(Mod(L)),n+1]\!]_b$的一条有穷执行 e 投影到函数调用、函数返回、调用刷出和返回刷出动作集合上, 得到的扩展历史为 eh, 并且在 e 的最后一个格局中所有进程的存储缓冲区都被清空, 那么我们称有穷执行 e 是$[\![Clt_f(Mod(L)),n+1]\!]_b$的扩展历史 eh 的标记见证. 引入标记见证的原因是: 这样的执行的开始格局和终止格局都无需考察存储缓冲区的内容, 方便完成从 TSO 内存模型下可线性化的限界版本到易失通道机器的控制状态可达问题的归约. 下面的引理将判定特定扩展历史是否存在这个问题归约到了判定特定的标记见证是否存在这个问题.

引理 4. *ehistory*($[\![L,n]\!]_{tt}$)上存在扩展历史 *eh*, 当且仅当$[\![Clt_f(Mod(L)),n+1]\!]_b$存在 *eh* 的一条标记见证.

引理 4 的简要证明如下: 引理的"当"方向由引理 1 和引理 2 显然可以得到. 对"仅当"方向, 根据引理 1 和引理 2, *ehistory*($[\![L,n]\!]_{tt}$)上存在扩展历史 *eh*, 当且仅当$[\![Clt_f(Mod(L)),n+1]\!]_b$上存在执行 *t*, 且 *t* 的扩展历史为 *eh*. 在 *t* 的最后一个动作之后, 我们令进程 P_1 到 P_n 都清空自己的存储缓冲区, 就得到了 *eh* 的一条标记见证.

令 M_i^{eh-f} 是通过对 M_i^{eh} 做如下更改得到的通道机器: 首先, 将除了刷出、*cas*、调用刷出和返回刷出之外的迁移都转化为τ迁移; 然后, 将每个 *cas*(j,x,d_1,d_2)迁移转化为 *flush*(j,x,d_2)迁移; 再将每个调用刷出 *flushCall*(j)迁移转化为 *flush*(j,z_f,d)迁移, 这里的 d 是 M_i^{eh} 产生 *flushCall*(j)迁移时刷出的对应 z_f 的项中的内存快照; 最后, 用类似的方法处理返回刷出迁移.

下面的引理说明: 给定 *k*-扩展历史 *eh* 和一对格局, 以这对格局为起点和终点的 *eh* 对应的标记见证的存在问题(根据引理 4, 即 *eh* 的存在问题)可以被归约到判断$\bigcap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{eh-f} \neq \varnothing$. 注意, 这里的机器并未采用易失通道. 这个引理的成立, 来自我们事实上用 z_w 的刷出之后紧邻的τ迁移对应了函数调用和函数返回迁移, 用 z_f 的"刷出"对应了调用刷出和返回刷出迁移. 这个引理的证明可以在本文的技术报告版本[29]中找到.

引理 5. 给定一条 *k*-扩展历史 *eh*. $[\![Clt_f(Mod(L)),n+1]\!]_b$ 上存在从$(p_{in},d_{init},u'_{init},\perp)$开始、到$(p_w,d_w,u'_{init},\perp)$结束的 *eh* 的标记见证, 当且仅当$\bigcap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{eh-f} \neq \varnothing$. 这里, p_{in} 将进程 P_1 到 P_n 映射到 in_{clt}, 且将进程 P_{n+1} 映射到 q_{wit}. u'_{init} 将进程 P_1 到 P_{n+1} 映射到ε. 对 $1\leqslant i\leqslant n+1$, $q_i=(p_{in}(i),d_{init},d_{init},q_{is},\perp)$, 且 $q_i'=(p_w(i),d_w,d_w,q_{ends},\perp)$.

令 M_i^{eh-w} 是通过对 M_i^{eh} 做如下更改得到的通道机器: 首先, 将除了写和 *cas* 之外的迁移都转化为τ迁移; 然后, 将每个 *cas*(j,x,d_1,d_2)迁移转化为 *write*(j,x,d_2)迁移. 由于非易失通道不丢失内容, 因此, M_i^{eh-f} 的刷出动作序列和 M_i^{eh-w} 的写动作序列一致. 进而, 引理 5 在将 M_i^{eh-f} 替换为 M_i^{eh-w} 时仍成立.

当把 M_i^{eh-w} 视为易失通道机器时, 为了保证不丢失通道内对应扩展历史 *eh* 的 z_f 和 z_w, 通道中的字母表 Σ_{s_1} 中的元素不应丢失. 值得注意的是: 虽然 M_i^{eh-w} 中的函数调用、函数返回、调用刷出和返回刷出迁移被转化为了τ迁移, 但 M_i^{eh} 的状态中的第 4 个元组, 即自动机 A_{eh} 的状态, 仍在记录扩展历史, 记录的扩展历史来自于通道中缓存的 z_f 和 z_w. 为了保证 M_i^{eh-w} 的读动作总能成功读到对相应内存单元的最新的写, 通道中的字母表 Σ_{s_2} 中的元素不应丢失, 因此, 强符号集合的向量为 $S=(\Sigma_{s_1},\Sigma_{s_2})$. 由于扩展历史 *eh* 的长度为 *k*, 通道里的对应扩展历史的项(z_f 和 z_w)的数目不超过 *k*. 每个通道里字母表 Σ_{s_2} 中的元素数目不会超过$|X_L|$, 因此, 强符号约束为(S,K), 其中, $K=\langle k,|X_L|\rangle$. 下面的引理说明了引理 5 在将通道机器 M_i^{eh-f} 替换为(S,K)-易失通道机器 M_i^{eh-w} 时仍成立. 这个引理的证明可以在本文的技术报告版本[29]中找到.

引理 6. 给定一条 *k*-扩展历史 *eh*. $[\![Clt_f(Mod(L)),n+1]\!]_b$ 上存在从$(p_{in},d_{init},u'_{init},\perp)$开始、到$(p_w,d_w,u'_{init},\perp)$结束的 *eh* 的标记见证, 当且仅当$\bigcap_{i=1}^{n+1} LT_{(q_i,q_i')}^{(S,K_1)} M_i^{eh-f} \neq \varnothing$. 这里, p_{in} 将进程 P_1 到 P_n 映射到 in_{clt}, 且将进程 P_{n+1} 映射到 q_{wit}. u'_{init} 将进程 P_1 到 P_{n+1} 映射到ε. 对 $1\leqslant i\leqslant n+1$, $q_i=(p_{in}(i),d_{init},d_{init},q_{is},\perp)$, 且 $q_i'=(p_w(i),d_w,d_w,q_{ends},\perp)$.

根据引理 3, $\bigcap_{i=1}^{n+1} LT_{(q_i,q_i')}^{(S,K_1)} M_i^{eh-w} \neq \varnothing$ 成立, 当且仅当 $LT_{(p,p')}^{(S,K_1)} M_1^{eh-w} \otimes \ldots \otimes M_{n+1}^{eh-w} \neq \varnothing$ 成立. 后者正是(S,K)-易失通道机器的控制状态可达问题, 是可判定的. 这里, $p=(q_1,\ldots,q_{n+1})$, 且 $p'=(q_1',\ldots,q_{n+1}')$. 引理 6 中的 p_w 和 d_w 的数目是有穷的. 通过枚举所有的 p_w 和 d_w 并多次使用引理 6, 可以判定$[\![Clt_f(Mod(L)),n+1]\!]_b$上是否存在 *eh* 的标记见证. 根据引理 4, 这个方法判定了 *eh* 是否是$[\![L,n]\!]_{tt}$上的 *k*-扩展历史. 根据第 4 节的讨论, 我们可以得到 TSO 内存模型下可线性化的 3 种限界版本是可判定的, 如下面的定理所述.

定理 7. 给定并发数据结构 *L* 和 *L′*、顺序规约 *Spec* 以及正整数 *n*, *n* 进程下 *L′*是否 *k*-限界 TSO-to-TSO 线性化了 *L* 是可判定的, *n* 进程下 *L′*是否 *k*-限界 TSO-to-SC 线性化了 *L* 是可判定的, *n* 进程下 *L* 是否 *k*-限界 TSO 可线性化到 *Spec* 是可判定的.

我们使用简单通道机器来称呼只使用迁移标号ε、且迁移规则中不使用正则卫士表达式和替换的通道机器. 简单通道机器的可达问题[20]是指: 给定简单通道机器 M、格局 s_1 和 s_2(一个格局包含控制状态和通道内

容), 判断作为易失通道机器的 M 是否有一条从 s_1 和 s_2 的有穷执行. 文献[20]证明了简单通道机器的可达问题的复杂度在递归函数的 Fast-Growing 层级[19] $\mathfrak{F}_{\omega^\omega}$ 中.

为了证明的方便, 我们使用单通道简单通道机器, 为此需要证明简单通道机器的可达问题与单通道的简单通道机器的可达问题可以相互归约.

- 一方面, 后者是前者的一个子问题;
- 另一方面, 我们通过如下方式将前者归约为后者: 给定简单通道机器 M, 可以构造一个单通道简单通道机器 M'. M'的通道内容就是使用分隔符分隔开的 M 的各个通道的内容. M 的一步迁移中对通道内容的修改被 M'上若干步迁移所模拟, 这些迁移读取整个通道的内容, 并分别修改"M 的每个通道"的内容(根据分隔符确定是哪个通道). 因此, 单通道简单通道机器的可达问题与简单通道机器的可达问题有着相同的复杂度. 通过类似的思想, 可以根据易失通道机器 M 构造单通道简单通道机器 M'. 为了 M 的一步迁移中对通道内容的修改被 M'上若干步迁移所模拟, 这些迁移读取整个通道的内容, 使用正则卫士表达式进行判断, 进行可能的元素替换, 并分别修改"M 的每个通道"的内容. 因此可以证明, 易失通道机器的控制状态可达问题和单通道的简单通道机器的可达问题可以互相归约.

由于 TSO 内存模型下可线性化的 3 种限界版本的验证问题被归约为易失通道机器的控制状态可达问题, 因此可以看到, TSO 内存模型下可线性化的 3 种限界版本的验证问题可以被归约到单通道的简单通道机器的可达问题. 注意: 由于易失通道机器 M_i^{oh} 能较为自然地建模进程 i 的执行, 因此在可线性化的 3 种限界版本的可判定性证明中, 我们将其归约到易失通道系统的控制状态可达, 而不是单通道的简单通道机器的可达问题.

采用类似于文献[17]的思想, 我们构造了并发数据结构 $L^M_{(s_1,s_2)}$, 它使用运行在两个进程上的两个函数 M_1 和 M_2 的合作来模拟单通道简单通道机器 M 使用易失通道时, 从 s_1 开始的迁移. 两个函数不断地读取来自因对方的存储缓冲区刷出而导致的更新, 并将读到的值写入自己的存储缓冲区(在必要时, 根据 M 的迁移规则修改读到的值, 并将新的值写入自己的存储缓冲区). 通过这种方法, 模拟了通道及通道机器的迁移. 由于 TSO 内存模型下, 一个进程的刷出未必会被另一个进程读到, 这导致模拟的通道可能"丢失"一些内容, 即模拟了易失通道. 只有当模拟的通道机器的执行到达 s_2 之后, $L^M_{(s_1,s_2)}$ 的两个函数才可以返回. 通过这种方法, 我们将单通道简单通道机器 M 的可达问题, 归约到是否某条 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的扩展历史包含函数返回动作. 并发数据结构 $L^M_{(s_1,s_2)}$ 的构造以及这个归约的证明, 可以在本文的技术报告版本[29]中找到.

下面叙述如何通过将单通道简单通道机器的可达问题, 归约到判定 $L^M_{(s_1,s_2)}$ 是否符合 TSO 内存模型下可线性化的 3 种限界版本的方法. 具体的证明可以在本文的技术报告版本[29]中找到.

- k-限界 TSO-to-TSO 可线性化的复杂度

令并发数据结构 L_{pend} 包含 M_1 和 M_2 两个函数, 且这两个函数从不返回. 一种实现方法为 M_1 和 M_2 内只包含一条 while(true);语句. 当 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 中存在一条包含至少一个函数返回动作的执行时, 令 t 为这样的执行, 且不妨设这个函数返回动作发生在进程 P_1. 基于 t, 我们构造另一个执行 t'如下: 如果进程 P_1 尚未做调用刷出, 则进程 P_1 的存储缓冲区内缓存着(z_f,*call*), 将其刷出并进行一次调用刷出动作; 由于需要 M_1 和 M_2 的合作来模拟 M 的行为, 进程 P_2 一定有至少一次函数调用; 如果进程 P_2 尚未做调用刷出, 则进程 P_2 的存储缓冲区内缓存着(z_f,*call*), 将其刷出并进行一次调用刷出动作. 执行 t'显然属于 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 且至少包含 5 个函数调用、函数返回、调用刷出或返回刷出动作. 显然, 执行 t'的扩展历史不属于 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的扩展历史集合.

可以证明: $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 存在一条包含至少一个函数返回动作的执行, 当且仅当 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的扩展历史集合不是 $[\![L_{pend},2]\!]_{tt}$ 的扩展历史集合的子集, 当且仅当 2 进程下 L_{pend} 并未 TSO-to-TSO 线性化了 $L^M_{(s_1,s_2)}$. 由于 $[\![L_{pend},2]\!]_{tt}$ 的每条扩展历史最多包含 4 个动作, 且执行 t'的扩展历史包含至少 5 个动作, 因此在考察 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的扩展历史集合时, 只需要考察长度不超过 5 的扩展历史集合即可. 因此, 我们将易失通道机器 M 上的可达

问题归约到判定 2 进程下 L_{pend} 是 5-限界 TSO-to-TSO 线性化了 $L^M_{(s_1,s_2)}$.

- k-限界 TSO-to-SC 可线性化的复杂度

当 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 中存在一条包含至少一个函数返回动作的执行时, 令 t 为这样的执行, 且不妨设这个函数返回动作发生在进程 P_1. 由于需要 M_1 和 M_2 的合作来模拟 M 的行为, 进程 P_2 一定有至少一次函数调用. 执行 t 至少包含 3 个函数调用或函数返回动作. 显然, 这样的执行的历史不属于 $[\![L_{pend},2]\!]_{tt}$ 的历史集合.

可以证明: $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 存在一条包含至少一个函数返回动作的执行, 当且仅当 2 进程下 L_{pend} 并未 TSO-to-SC 线性化了 $L^M_{(s_1,s_2)}$. 由于 $[\![L_{pend},2]\!]_{tt}$ 的每条历史最多包含 2 个动作, 且执行 t 的历史包含至少 3 个元素, 因此在考察 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的历史集合时, 只需要考察长度不超过 3 的历史集合即可. 因此, 我们将易失通道机器 M 上的可达问题归约到判定 2 进程下 L_{pend} 是否 3-限界 TSO-to-SC 线性化了 $L^M_{(s_1,s_2)}$.

- k-限界 TSO 可线性化的复杂度

在并发数据结构 $L^M_{(s_1,s_2)}$ 的构造中, 要求一旦 M_1 或者 M_2 可以返回, 则其会修改一个 *flag*, 使得之后不管在哪个进程上的 M_1 和 M_2 都直接返回. 令顺序规约 *Spec* 为包含不超过两个 M_1 或 M_2 操作的顺序历史集合. 显然, S 是正则的, 且可以通过枚举构造. 当 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 中存在一条包含至少一个函数返回动作的执行时, 令 t 为这样的执行, 且不妨设这个函数返回动作发生在进程 P_1. 基于 t, 我们构造另一个执行 t' 如下: 此时, M_1 和 M_2 都变成了实质上的空函数. 如果进程 P_1 尚未做调用刷出或返回刷出, 则清空进程 P_1 的存储缓冲区, 并做调用刷出或返回刷出. 由于需要 M_1 和 M_2 的合作来模拟 M 的行为, 进程 P_2 一定有至少一次函数调用. 如果进程 P_2 的函数还未返回, 则在进程 P_2 执行一次函数返回动作. 之后, 清空进程 P_2 的存储缓冲区, 并做调用刷出或返回刷出. 最后, 在进程 P_1 上调用一个函数并返回, 并做调用刷出和返回刷出. 执行 t' 显然属于 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 且至少包含 6 个函数调用或返回刷出动作, 对应至少 3 个操作. 显然, *Spec* 中的顺序历史包含的操作少于 3 个.

可以证明: $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 存在一条包含至少一个函数返回动作的执行, 当且仅当 2 进程下 $L^M_{(s_1,s_2)}$ 并未 TSO 可线性化到 *Spec*. 由于 *Spec* 中的每个历史包含不超过两个操作, 且执行 t' 对应至少 3 个操作, 包含至少 6 个函数调用或返回刷出动作, 因此在考察 $[\![L^M_{(s_1,s_2)},2]\!]_{tt}$ 的函数调用和返回刷出动作序列时, 只需要考察长度不超过 6 的序列集合即可. 因此, 我们将易失通道机器 M 上的可达问题归约到判定 2 进程下 $L^M_{(s_1,s_2)}$ 是否 6-限界 TSO 可线性化到 S.

因此, 我们将 TSO 内存模型下可线性化的 3 种限界版本的验证问题归约到单通道的简单通道机器的可达问题, 又将后者归约到了前者. 根据以上讨论, 得到了如下定理, 它说明 TSO 内存模型下可线性化的 3 种限界版本的复杂度都在递归函数的 Fast-Growing 层级 $\mathfrak{F}_{\omega^\omega}$ 中.

定理 8. 在 n 进程下, 检测 k-限界 TSO-to-TSO 可线性化、k-限界 TSO-to-SC 可线性化和 k-限界 TSO 可线性化的复杂度都在递归函数的 Fast-Growing 层级 $\mathfrak{F}_{\omega^\omega}$ 中.

8 结论和未来工作

据我们所知, TSO-to-TSO 可线性化、TSO-to-SC 可线性化和 TSO 可线性化是文献中针对 TSO 内存模型提出的全部可线性化变种. 本文研究了 k-限界 TSO-to-TSO 可线性化、k-限界 TSO-to-SC 可线性化和 k-限界 TSO 可线性化的可判定性. TSO 内存模型下可线性化的 3 种限界版本都是非平凡的, 因为写动作的数目不限界, 因此使用的存储缓冲区大小不限界, 进而其操作语义是无穷状态迁移系统. 我们证明了 TSO 内存模型下可线性化的这 3 种限界版本都是可判定的. 我们的工作解决了 TSO 内存模型下限界可线性化的可判定性问题.

本文将并发数据结构以及顺序规约之间的 k-限界 TSO-to-TSO 可线性化、k-限界 TSO-to-SC 可线性化和 k-限界 TSO 可线性化的验证归约到 k-扩展历史集合之间的 TSO-to-TSO 可线性化的验证, 从而以一种统一的方式验证 TSO 内存模型下可线性化的 3 个限界版本. 验证的重点在于判定 TSO 内存模型下并发数据结构是

否有一条特定的扩展历史. 我们证明了这个问题是可判定的, 证明的方法是将其归约到已知可判定的易失通道机器的控制状态可达问题. 归约的难点在于: 已有的关联 TSO 内存模型下并发系统和易失通道机器的工作关注的问题是基于状态定义的, 而并发数据结构的正确性定义是基于函数调用、函数返回、调用刷出和返回刷出动作的, 这两个问题的粒度不同. 更加复杂的是, 函数调用和函数返回动作以一种复杂的方式同时影响控制状态和存储缓冲区. 我们的工作通过修改内存模型、并发数据结构、客户程序和操作语义, 使得内存单元 z_f 和 z_w 的动作序列对应了函数调用、函数返回、调用刷出和返回刷出的序列. 然后,将修改的操作语义上是否存在特定的扩展历史这个问题归约到了易失通道机器 $M_1^{eh-w} \otimes ... \otimes M_{n+1}^{eh-w}$ 的控制状态可达问题. 我们提出的这个归约方法将有助于验证松弛内存模型上的其他粗粒度性质.

进而证明了在 TSO 内存模型下判定可线性化的这 3 个限界版本的复杂度都在递归函数的 Fast-Growing 层级 $\mathfrak{F}_{\omega^\omega}$ 中. 通过证明单通道简单通道机器的可达问题这一已知的有着这样复杂度的问题和 TSO 内存模型下可线性化的这 3 个限界版本可以互相归约得到这个结论. 我们的工作展示了在 TSO 内存模型下对限界可线性化的验证具有理论可行性, 为 TSO 内存模型一致性验证的进一步研究打下了理论基础. 非限界版本的 TSO-to-SC 可线性化和 TSO 可线性化的可判定性问题目前仍是开放的. 作为未来工作, 我们准备研究非限界版本的 TSO-to-SC 可线性化和 TSO 可线性化的可判定性问题.

References:

[1] Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems, 1990, 12(3): 463−492.

[2] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers, 1979, 28(9): 690−691.

[3] Sewell P, Sarkar S, Owens S, *et al*. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM, 2010, 53(7): 89−97.

[4] Sarkar S, Sewell P, Alglave J, *et al*. Understanding POWER multiprocessors. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2011). New York: ACM, 2011. 175−186.

[5] Pulte C, Flur S, Deacon W, *et al*. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. Proceedings of the ACM on Programming Languages (PACMPL), 2018, 2(POPL): Article 19.

[6] Nienhuis K, Memarian K, Sewell P. An operational semantics for C/C++11 concurrency. In: Proc. of the ACM SIGPLAN Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). 2016. 111−128.

[7] Manson J, Pugh W, Adve SV. The Java memory model. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2005). New York: ACM, 2005. 378−391.

[8] Atig MF, Bouajjani A, Burckhardt S, *et al*. On the verification problem for weak memory models. In: Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2010). New York: ACM, 2010. 7−18.

[9] Owens S, Sarkar S, Sewell P. A better x86 memory model: x86-tso. In: Proc. of the TPHOLs. 2009. 391−407.

[10] Bouajjani A, Emmi M, Enea C, *et al*. Verifying concurrent programs against sequential specifications. In: Proc. of the 22nd European Conf. on Programming Languages and Systems (ESOP 2013). Berlin, Heidelberg: Springer-Verlag, 2013. 290−309.

[11] Burckhardt S, Gotsman A, Musuvathi M, *et al*. Concurrent library correctness on the TSO memory model. In: Seidl H, ed. Proc. of the 21st European Symp. on Programming (ESOP 2012). Tallinn: Springer, 2012. 87−107.

[12] Gotsman A, Musuvathi M, Yang HS. Show no weakness: Sequentially consistent specifications of TSO libraries. In: Proc. of the 26th Int'l Symp. on Distributed Computing (DISC 2012). 2012. 31−45.

[13] Derrick J, Smith G, Dongol B. Verifying linearizability on tso architectures. In: Albert E, Sekerinski E, eds. Proc. of the Integrated Formal Methods. Cham: Springer, 2014. 341−356.

[14] Batty M, Dodds M, Gotsman A. Library abstraction for C/C++ concurrency. In: Giacobazzi R, Cousot R, eds. Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2013). Rome: ACM, 2013. 235−248.

[15] Doherty S, Dongol B, Wehrheim H, *et al*. Making linearizability compositional for partially ordered executions. In: Proc. of the 14th Int'l Conf. on Integrated Formal Methods (IFM 2018). 2018. 110−129.

[16] Alur R, McMillan K, Peled D. Model-checking of correctness conditions for concurrent objects. In: Proc. of the 11th Annual IEEE Symp. on Logic in Computer Science (LICS'96). Washington: IEEE Computer Society, 1996. 219−228.

[17] Wang C, Lv Y, Wu P. TSO-to-TSO linearizability is undecidable. Acta Informatica, 2018, 55(8): 649−668.

[18] Wang C, Lv Y, Wu P. Bounded TSO-to-SC linearizability is decidable. In: Freivalds RM, Engels G, Catania B, eds. Proc. of the SOFSEM 2016. Berlin, Heidelberg: Springer, 2016. 404−417.

[19] Wainer SS. A classification of the ordinal recursive functions. Archive for Mathematical Logic, 1970, 13(3-4): 136−153.

[20] Chambart P, Schnoebelen P. The ordinal recursive complexity of lossy channel systems. In: Proc. of the 23rd Annual IEEE Symp. on Logic in Computer Science (LICS 2008). Pittsburgh: IEEE Computer Society, 2008. 205−216.

[21] Liang HJ, Feng XY, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. In: Field J, Hicks M, eds. Proc. of the 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2012). Philadelphia: ACM, 2012. 455−468.

[22] Liang HJ, Feng XY. Modular verification of linearizability with non-fixed linearization points. In: Boehm HJ, Flanagan C, eds. Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2013). Seattle: ACM, 2013. 459−470.

[23] Bouajjani A, Emmi M, Enea C, Hamza J. On reducing linearizability to state reachability. In: Halldorsson MM, Iwama K, Kobayashi N, Speckmann B, eds. Proc. of the 42nd Int'l Colloquium on Automata, Languages, and Programming (ICALP 2015). Kyoto: Springer, 2015. 95−107.

[24] Bouajjani A, Enea C, Wang C. Checking linearizability of concurrent priority queues. In: Meyer R, Nestmann U, eds. Proc. of the 28th Int'l Conf. on Concurrency Theory (CONCUR 2017). Berlin: Schloss Dagstuhl-Leibniz-Zentrum fur Informatik, 2017. 16:1−16:16.

[25] Liu Y, Chen W, Liu YA, *et al*. Verifying linearizability via optimized refinement checking. IEEE Trans. on Software Engineering, 2013, 39(7): 1018−1039.

[26] Derrick J, Smith G, Groves L, *et al*. A proof method for linearizability on TSO architectures. In: Proc. of the Provably Correct Systems. 2017. 61−91.

[27] Travkin O, Wehrheim H. Handling TSO in mechanized linearizability proofs. In: Yahav E, ed. Proc. of the Hardware and Software: Verification and Testing. Cham: Springer, 2014. 132−147.

[28] Atig MF. What is decidable under the TSO memory model? ACM SIGLOG News, 2020, 7(4): 4−19.

[29] Wang C, Lv Y, Wu P, *et al*. Bounded linearizability on tso memory model is decidable. Technical Report, ISCAS-SKLCS-21-01, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, 2021. http://lcs.ios.ac.cn/~lvyi/files/ISCAS-SKLCS-21-01.pdf

王超(1985－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化方法, 并发数据结构, 分布式数据类型.

吴鹏(1977－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为形式化方法, 并发测试, 机器学习.

吕毅(1972－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为并发理论, 形式化方法.

贾巧雯(1992－), 女, 博士生, 主要研究领域为并发系统, 可线性化验证.