arXiv:2304.03753v1 [cs.PL] 7 Apr 2023

带同步的响应式并行

STEFAN K. MULLER，伊利诺伊理工学院，美国KYLE SINGER，圣路易斯华盛顿大学，美国DEVYN TERRA KEENEY，伊利诺伊理工学院，美国ANDREW NETH，伊利诺伊理工学院，美国KUNAL AGRAWAL，圣路易斯华盛顿大学，美国I-TING ANGELINA LEE，圣路易斯华盛顿大学，美国UMUT A. ACAR，卡内基梅隆大学，美国

许多并发程序会为线程分配优先级，以提升响应性。然而，当优先级与互斥锁和条件变量等同步机制结合使用时，可能会导致优先级反转，即高优先级线程被低优先级线程延迟。互斥锁使用中的优先级反转很容易通过优先级继承等动态技术处理，但条件变量使用中的优先级反转尚未得到充分研究，且动态技术并不适用。

在这项工作中，我们结合静态技术和动态技术，防止使用互斥锁和条件变量的代码发生优先级反转。类型系统确保条件变量得到安全使用，即使动态技术会在运行时更改线程优先级，以消除使用互斥锁时出现的优先级反转。我们使用基于并行程序成本模型的优先级反转模型，证明了我们系统的可靠性。为了说明该类型系统具有实际可实现性，我们将其编码到 Rust 和 C++ 的类型系统中，并通过使用这些编码编写规模较大的案例研究，说明这些限制并不会带来过重的负担；其中包括将Memcached 对象服务器移植为使用我们的 C++ 实现。

1 引言

数十年来，软件应用程序一直使用并发，在多个线程上同时执行任务。在许多此类应用程序中，尤其是与用户或外部环境交互的应用程序，不同任务对资源和延迟有不同的要求。例如，在电子邮件客户端中，处理用户输入的事件循环必须频繁运行，以确保用户界面的响应性；而利用空闲周期压缩已存储邮件的后台线程则没有此类要求。许多多线程系统允许程序员为线程指定优先级，以满足这些要求：对延迟高度敏感的线程可能以高优先级运行，而敏感度较低的后台线程则以低优先级运行。

几乎从优先级被用于并发程序以来，程序员和研究人员就一直在观察优先级反转问题。虽然这一问题存在许多略有差异的表述，但其本质是：高优先级线程发现自己正在等待低优先级线程释放某些资源或建立某个条件。在大多数情况下，等待会很短，因此这种有界优先级反转有时并不被认为本身有害。然而，如果该场景中的低优先级线程被一个长时间运行的中等优先级线程抢占，就可能演变为无界 优先级反转，从而严重影响响应性，甚至完全阻止程序取得进展。因此，能够检测并防止优先级反转是很有价值的。

作者地址：Stefan K. Muller，smuller2@iit.edu 伊利诺伊理工学院，美国；Kyle Singer，kdsinger@wustl.edu 圣路易斯华盛顿大学，美国；Devyn Terra Keeney，dkeeney2@hawk.iit.edu 伊利诺伊理工学院，美国；Andrew Neth，aneth@hawk.iit.edu 伊利诺伊理工学院，美国；Kunal Agrawal，kunal@wustl.edu 圣路易斯华盛顿大学，美国；I-Ting Angelina Lee，angelee@wustl.edu 圣路易斯华盛顿大学，美国；Umut A. Acar，umut@cmu.edu 卡内基梅隆大学，美国。

当优先级反转是由对简单同步原语（如锁或互斥锁）的竞争引起时，可以使用诸如 优先级继承 或优先级上限 [Sha等人提出的 1990], 动态机制来修复，这两种机制都会暂时提高低优先级线程的优先级，使其能够运行并释放资源。然而，在条件变量的语境下，人们对优先级反转问题的理解要少得多。条件变量是一种强大且通用的同步机制，可用于编码许多其他机制，例如信号量和监视器 [Hansen 1973、1975；Hoare 1974]。事实上，不可能使用类似于优先级继承或优先级上限的动态技术来检测和防止由条件变量引起的优先级反转。要了解其原因，请考虑一个由主线程 *A*组成的程序，该主线程会生成线程 *B* 并（过了一段时间后） *C*。如果线程 *B* 等待条件变量（该操作会阻塞，直到另一个线程执行相应的信号操作），那么线程调度器无法知道哪个线程将执行信号操作：事实上，在执行“等待”时，线程 *C* 甚至可能尚未生成。因此，甚至不可能即时确定这种阻塞是否构成优先级反转。

本文考虑解决这一问题的静态方法：我们的解决方案是一种类型系统，能够可靠地判定是否可能发生优先级反转。随后可以利用这一信息，拒绝存在潜在优先级反转的程序，或者向程序员发出警告，指出代码的特定部分可能需要进一步推理和/或调试，以确保不会发生优先级反转。我们在一种命令式演算的背景下研究该类型系统，该演算具有静态语义和动态语义，可处理条件变量和锁，并结合静态技术与动态技术来防止优先级反转：对于互斥锁，我们采用动态优先级天花板协议；对于条件变量，我们采用静态技术。

类型系统为每个互斥锁和条件变量静态关联一个优先级。对于互斥锁，这是优先级上限——它可能被获取时的最高优先级。对于条件变量，这是一个高于（或等于）可能在其上等待的线程、且低于（或等于）可能向其发信号的线程的优先级。对于条件变量而言，这样的优先级存在这一事实意味着，任何信号线程的优先级都将高于任何等待线程，这看起来似乎排除了优先级反转。然而，这还不够：考虑上面所述的三线程场景，其中线程 *B* and *C* 具有高优先级，但 *A* 具有低优先级： *B* 必须等待低优先级线程 *A*创建线程 *C*，后者将发信号并唤醒 *B*。使用该条件变量的两个线程都具有高优先级，但仍然存在优先级反转。我们通过引入条件变量的所有权概念 [Ahmed等人 2007年；Boyland2003年；Crary 等人1999年；Grossman 等人 2002年；Smith 等人 2000] ，并允许代码在某些条件下提升条件变量的优先级，或提高其优先级，来防止此类反转。在上述示例中，线程 *A* 只能将条件变量提升为高优先级，从而允许线程 *B* 在其上等待，在创建线程 *C*之后，进而避免了优先级反转。类型系统不会静态限制互斥锁的使用，除了确保经注释的优先级上限是正确的之外，但它确实确保，在混合使用互斥锁和条件变量的代码中，使用优先级天花板协议不会导致运行时类型错误。

要证明类型系统能够可靠地检测优先级反转，需要一个足够丰富的优先级反转形式模型，以描述上述三线程优先级反转等场景。Babaoğlu等人 [1993], 提出的形式化方法基于线程在时间快照中的关系，无法对等待线程与尚未创建的信号线程之间的反转进行编码。相反，我们建立在一项近期工作 [Muller等人 2017, 2018, 2020] 的基础上，该工作将带有优先级的并行程序表示为有向无环图（DAG），并表明，如果一个有向无环图满足某些“良构性”条件，则相应的程序可以通过一种简单的调度原则高效地进行调度，该原则会贪婪地遵循优先级。这给出了一个根据优先级反转对性能产生的影响来定义优先级反转的方法：优先级反转是线程之间的任何交互

带同步的响应式并行

```
1 void <低>f(条件变量 cv, int *result){ 2 ... 3 *result =...; 4 信号(cv); 5 } 6 void <高>main() { 7 条件变量 cv = new 条件变量(); 8 int result = 0; 9 生成<低>(f(条件变量, &result)); 10 ... 11 等待（条件变量）(cv); 12 }
```

图 1. 优先级反转示例。

```
1 void <低>f(CV<高> cv, int*result) { 2 ... 3 *result =...; 4 signal(cv); //类型错误 5 } 6 void <高>main() { 7 CV<高> cv = new CV<高>(); 8 int result = 0; 9 生成<低>(f(CV, &result)); 10 ... 11 等待(cv); 12 }
```

图 2. 优先级反转现在会导致类型错误。

这种反转会使高优先级线程因低优先级线程而延迟，而我们可以确信，对于良构的有向无环图，不会发生这种延迟。此前的工作也研究了用于防止优先级反转的类型系统，但仅适用于在线程完成时进行同步的程序；与互斥锁和条件变量所允许的模型相比，这种模型弱得多。我们扩展了有向无环图模型、良构性定义和调度证明，以表示使用互斥锁和条件变量的并发程序，并证明由良类型程序产生的任何有向无环图都是良构的，因此不会发生优先级反转。

尽管功能强大，我们在本文中提出的类型系统采用了一些成熟的思想，只是以新颖的方式加以运用。事实上，我们能够利用 C++ 和 Rust 类型系统的高级特性，对其限制进行近似编码（但存在一些局限）。为了证明类型系统的限制不会过度妨碍生产力，我们利用这些编码开发了规模可观的案例研究程序，其中包括一个大型的真实世界交互式应用程序——Memcached 对象缓存服务器 [Mem 2009] (v1.5.13)，它包含约 20,100 行 C 代码。

我们首先通过示例概述类型系统的关键思想（第 2 节），随后正式介绍类型系统（第3节）和成本模型（第 4 节）。在第 5 节中，我们定义核心演算的动态语义，包括优先级天花板协议的支持机制，并证明类型安全和良构性（不存在优先级反转）。随后，我们讨论 C++ 和 Rust 的实现及案例研究，介绍相关工作，并给出结论。

2 概述与示例

本节通过若干示例，介绍防止涉及互斥锁和条件变量的优先级反转问题的复杂性，并阐述我们提出的方法背后的主要思想；我们将在第3节对该方法进行形式化说明。

2.1 条件变量

我们考虑条件变量的三种操作（除了创建新条件变量的操作）：等待、信号和提升。等待操作会使调用线程在条件变量上阻塞，而信号操作会唤醒当前阻塞在条件变量上的一个线程。[1] 提升操作将在本节后面讨论。图 1 中的代码使用一个异步线程运行函数 f，并将其结果写入引用 result，随后

[1] We 我们将在本文后面讨论如何将唤醒所有阻塞线程的广播操作作为一种直接的扩展加入。

该线程向条件变量 cv 发送信号，以表明结果已可用。[2] 主线程随后执行其他工作；当需要异步计算的结果时，它会在该条件变量上等待。

遗憾的是，这段代码存在优先级反转：高优先级主线程等待正在运行 f 的低优先级线程。与先前工作 [Muller 等人 2020], 中的做法一样，仅凭线程上的优先级注释无法静态检测这种优先级反转，因为调用 wait 时，类型系统并不知道哪个线程会调用 signal。因此，我们为条件变量本身指定优先级，并限制线程对条件变量的使用。首先，我们要求：

（1）要等待优先级为 ρ 的条件变量，线程的优先级必须小于或等于 ρ；（2）要向优先级为 ρ 的条件变量发信号，线程的优先级必须大于或等于 ρ。

这就通过传递性确保任何信号线程的优先级都高于任何等待线程。上述两个条件成为我们在第3节中形式化的类型系统所强制执行的前两项限制；随着我们考察更多优先级反转示例，还会继续增加限制。

在图 2 中，我们将条件变量 cv 定义为高优先级，因为主线程会在其上等待。此时，优先级反转表现为类型错误，因为执行 f 的低优先级线程无法向高优先级条件变量发送信号。请注意，将 cv 切换为低优先级并不能解决问题，因为这样一来，高优先级的主线程就会在低优先级的条件变量上等待。

一种微妙的优先级反转。 类型限制的大部分复杂性源于一种特定的优先级逆向分析模式，我们通过经典“生产者-消费者”模式的一个实例来说明。图 3 展示了伪代码：主线程以低优先级运行，并生成一个将消息放入缓冲区的“生产者”线程，以及一个从缓冲区取出并处理消息的“消费者”线程。生产者和消费者都以高优先级运行。如果消费者领先于生产者并发现缓冲区为空，我们希望它休眠，直到缓冲区变为非空。[3] 为此，两个线程还共享一个条件变量 cv。由于高优先级线程会等待该变量，它必须具有高优先级。如果消费者发现缓冲区为空，就会等待 cv。生产者向缓冲区添加元素时会向 cv 发送信号，从而在消费者被阻塞时将其唤醒。

这个代码示例满足约束线程和条件变量优先级的条件 1 和 2：在该示例中，所有相关线程和条件变量的优先级都很高。然而，这段代码仍然包含一个微妙的优先级反转。考虑在单处理器上以分时方式运行该程序。当主线程生成消费者时，消费者会立即开始运行（因为它的优先级很高），并一直运行到发现缓冲区为空，然后等待条件变量 cv。此时，高优先级的消费者等待的并非高优先级的生产者，而是低优先级的主线程；主线程仍需运行才能创建生产者。在我们的分时系统中，如果一个运行时间很长的中优先级线程（伪代码中未显示）抢占了低优先级主线程，那么这种等待实际上可能变得没有时限。

问题在于，低优先级的主线程持有高优先级的条件变量 cv。发生这种情况时，等待该条件变量的线程可能会被持有它的低优先级线程阻塞。为避免这种情况，我们增加一项限制（稍后将更正式地重述），用于约束线程与条件变量之间的交互：

(3)（草稿）如果一个线程的优先级为 ρ，那么它及其后代线程不得向优先级高于 ρ 的任何条件变量发送信号。

[2]这可以看作是对 *future* 或 *promise* 的编码，这是一种常用的并行性表达机制。[3]为简单起见，并为了突出我们类型系统的新颖之处，我们假设缓冲区是无界的，因此生产者永远不需要阻塞。我们的系统也能表示经典情形：缓冲区满时生产者会阻塞。

带同步的响应式并行

```
1 void <高>prod(条件变量<高> cv, 缓冲区 *buf){ 2 While(true) { 3 ... 4 append(buf,x); 5 signal(cv); 6 } 7 } 8 9 void <高>cons(条件变量<高> cv, 缓冲区 *buf){ 10 While(true) { 11 if empty(buffer) { 12 wait(cv); 13 } 14 x = pop(buf); 15 ... 16 } 17 } 18 19 void <低>main() { 20 条件变量<高>cv = new CV(); 21 缓冲区 *buf; 22 23 生成<高>(cons(cv, buf)); 24 生成<高>(prod(cv, buf)); // 类型不正确 25 }
```

图 3. 初始尝试 对生产者-消费者程序的尝试，其中 其优先级为 版本（在我们的完整系统中还会产生类型错误）。

```
1 void <高>prod(CV<低>cv, Buffer *buf) { 2 while(true) { 3 ... 4 append(buf, x); 5 signal(cv); 6 } 7 } 8 9 void <低>main() { 10 CV<低> cv = newCV(); 11 Buffer *buf; 12 13 生成<高>(prod(cv, buf)); 14 CV<高> cv2 =提升<高>(cv); 15 // 现在再生成 prod 将导致类型错误 16 生成<高>(cons(cv2, buf)); 17 }
```

Fig. 4. 消费者者例，没有类型错误或优先级反转。

在图 3 的示例中，生成生产者的操作存在类型错误，因为低优先级主线程试图将高优先级条件变量传递给一个打算向其发送信号的线程（稍后会清楚为什么类型错误出现在生成操作上，而不是发送信号操作上）。

看起来我们现在陷入了僵局：条件变量的优先级必须是高（因为高优先级消费者正在等待它），但这样一来，低优先级主线程就无法传递它。为了取得进展，我们允许线程在特定条件下通过提高条件变量的优先级来提升 条件变量。请注意，当低优先级线程提升条件变量时，它会限制自己对该条件变量的操作权限，例如，它将无法再向其发送信号。图 4 展示了在生产者-消费者示例中，主线程如何提升条件变量。此时，条件变量最初的优先级为低，并以此优先级传递给生产者（这是可接受的，因为高优先级生产者可以向低优先级条件变量发送信号）。生成生产者后，主线程将条件变量提升到高优先级。promote 操作会返回一个新句柄 cv2，它与 cv 指向同一个底层条件变量，但其类型优先级为高。该句柄会传递给需要高优先级条件变量的消费者。请注意，这里生成线程的顺序很重要：将条件变量提升到高优先级后，任何会向条件变量发送信号的线程 若在此时生成，都会因限制 3 而产生类型错误。这可确保此时无法生成生产者线程，从而避免优先级反转和类型检查错误。

promote 操作。 在讨论使用和提升条件变量时的一整套类型限制之前，我们先概述条件变量 API 的限制，以及使用它的常见模式。

首先，发出条件变量信号的线程必须比等待该条件变量的线程具有更高的优先级。在我们的案例研究中，条件变量有许多用法，其中同一线程会在不同时间发出条件变量信号并等待条件变量（例如，程序中的同一线程可能在不同时间分别充当生产者和消费者）。在这种情况下，类型系统会强制所有使用该条件变量的线程具有相同的优先级。还有一种模式是，发信号的线程具有严格更高的优先级（例如，服务器应用程序中的高优先级交互线程以生产者-消费者的方式向低优先级后台线程发送日志消息）。

第二个需要关注的问题是使用条件变量的线程生成时的顺序和条件。必须对其进行控制，以避免图3 所示的优先级反转：低优先级线程生成消费者，而消费者随后阻塞，等待生产者被生成。如果所有使用条件变量的线程都处于相同的优先级 ρ，并且所有这些线程都调用 等待 和信号，那么这些线程都应由一个优先级与它们相同或更高的线程生成（一个常见错误是从低优先级的初始线程生成它们）。一种不涉及提升的解决方案是生成一个优先级为 ρ （或更高）的临时线程，由它生成所有使用条件变量的线程。如果条件变量的使用者分为高优先级的“生产者”和低优先级的“消费者”，则可以像图 4 中那样，以较低的优先级构造条件变量，生成生产者，然后提升条件变量并生成消费者。

类型限制详解。现在，我们将更详细地讨论此 API 的类型限制。对于大多数条件变量 API 用户来说，并不需要完全理解这些细节，但本讨论可作为形式化类型系统的引入。在继续之前，我们还必须讨论提升的另一项重要特性。假设在图 4 中，生产者线程改为以低优先级运行。这仍符合目前讨论过的所有限制：生产者会以自身的优先级向条件变量发送信号，而主线程仍可以将条件变量传递给生产者。当然，这段代码会产生优先级反转，因为高优先级的消费者会等待低优先级的生产者。为避免这种情况，我们进一步加强限制：如果任何并发运行的线程可能以低于 ρ 的优先级向条件变量发送信号，则不允许将该条件变量提升至优先级 ρ。

为了使其更加形式化，我们为条件变量赋予一种所有权概念（例如， [Ahmed等人 2007年; Boyland2003年；Crary 等人1999年；Grossman等人2002年；Smith 等人 2000]）。每个线程在每个优先级上都对某个条件变量拥有一定程度的所有权：从概念上讲，这些级别是“无”、“独占”和“共享”。有一个不变式：如果一个线程在优先级 ρ上独占某个条件变量，则没有其他线程会在 ρ 上独占或共享它（尽管另一个线程可能在不同的优先级 ρ'上独占或共享它）。只要没有线程在某个优先级上独占某个条件变量，任意数量的线程都可以在该优先级上共享它。现在，我们可以用所有权来重新表述约束条件变量和线程的规则（限制条件 1 不变）：

(1)要等待优先级为 ρ的条件变量，线程的优先级必须小于或等于 ρ；(2)要向优先级为 ρ 的条件变量发送信号，线程必须以该线程的优先级拥有或共享该条件变量。(3) 要将任意优先级的条件变量的所有权传递给另一个线程，线程必须以其自身的优先级拥有或共享该条件变量。(4) 任何线程都不得以低于某条件变量自身优先级的优先级拥有或共享该条件变量。(5) 要将条件变量提升至优先级 ρ，线程必须在低于 ρ的每个优先级上拥有 该条件变量。

带同步的响应式并行

```
互斥锁 mut;
void  <低>f() {
   with (mut) { ... }
}
```

```
void  <高>g() {
   with (mut) { ... }
}
```

```
void  <中>  h() {
   while (1) { ... }
}
```

图 5. 三个线程和一个互斥锁导致的优先级反转。

许多重要事实都可以直接从这些限制中推导出来。首先，这些限制防止了上述假设的优先级逆向分析：如果主线程将 cv从低提升到高，那么它必须（根据限制 5）在低优先级下拥有 cv；而如果已经生成的低优先级生产者向其发送信号，这一点是不可能的（根据限制1 以及所有权的定义）。其次，重述后的限制2 与限制4一起意味着原始的限制2：如果一个线程的优先级低于条件变量的优先级，那么它不能在线程优先级下拥有或共享该条件变量，因此也不能向其发送信号。第三，限制 4 直接表明，将条件变量提升到优先级 ρ 后，执行提升操作的线程还必须放弃在所有更低优先级下的所有权。最后，限制3的形式化表述意味着前文给出的非正式版本，因为线程无法在自身（较低）的优先级下拥有或共享更高优先级的条件变量，因此也无法以有意义的方式“持有”它，即将其传递给可能向其发送信号的其他线程。

2.2 互斥锁

互斥锁用于在代码临界区中实现互斥，也就是说，确保同一时刻只有一个线程执行临界区。本文所考虑的语言使用带有 $v\ s$ 的语法结构来包裹临界区。如果 v 是一个互斥锁，此结构会尝试获取该互斥锁；若获取成功，则执行 s，确保在执行 s 期间没有其他线程能够获取 v。执行完成后，它会释放该互斥锁。如果 v 引用的互斥锁已被另一个线程持有，此结构会阻塞，直到能够获取该互斥锁，然后继续执行 s。[4]

互斥锁导致优先级反转的经典情形如图 5 所示。假设低优先级线程先运行并获取互斥锁 mut。随后，高优先级线程运行并尝试获取 mut，但被阻塞。接着，中优先级线程抢占低优先级线程，使其长时间甚至无限期无法运行，从而延迟了被其阻塞的高优先级线程。（请注意，根据本文采用的定义，即使没有中优先级线程，优先级反转仍然存在；但若没有该线程，对延迟的影响将受低优先级线程临界区长度的限制。）

正如我们对条件变量所做的那样，我们可以对互斥锁的使用添加静态限制，以防止优先级反转。此类限制将要求任何尝试获取给定互斥锁的线程都必须具有单一优先级 $\rho_{\mathtt{mut}}$ 专属于该互斥锁（要理解为什么这是必要的，可以考虑两个优先级为 A 和B 的线程可能会争用一个互斥锁：如果低优先级线程获取成功，高优先级线程正在等待它，就会发生优先级反转；如果没有关于哪个线程将成功的静态保证，A 和 B 就必须是相同的优先级）。这并不像乍看之下那么严苛——程序员可以确定，对于每个互斥锁，线程可能尝试获取它时所处的最高优先级（通常称为优先级上限），以及任何临界区

[4]在许多较底层的语言中，这相当于 lock v; s; unlock v。我们的语法确保锁定与解锁成对出现，并且互斥锁始终由锁定它的线程解锁。这种语法还使类型检查器能够查看临界区的整个范围，这对我们的类型系统至关重要。

```
互斥锁<高>mut;
条件变量<低>cv;
void <高>生产者 () {
  while(true) {
    . . .
    with(mut) {
      append(buf, x)
      signal(cv)
    }
  }
}
```

```
void <低>消费者 () {
  While(true) {
    with(mut) {
      While (empty(缓冲区))
        等待（条件变量）(cv);      // 类型错误
      x = pop(buf);
    }
    . . .
  }
}
```

图 6. 使用互斥锁和条件变量的、更贴近实际的生产者-消费者问题版本。

将以该互斥锁的优先级上限运行。[5] 这与 Lampson 和 Redell[1980] 提出的方案类似。

然而，大多数实现会采用多种更高效的方法之一来动态防止优先级反转，本文也对其中一种方法进行了建模。与上文提出的解决方案类似，我们将这种方法称为优先级天花板协议，它为每个互斥锁关联一个优先级上限，即可能尝试获取该互斥锁的线程中的最高优先级。创建互斥锁时会指定此优先级。较低优先级的线程 T 可以获取互斥锁，但如果较高优先级的线程在 T 的临界区内尝试获取该互斥锁，T就会被动态提升到优先级上限，并在其临界区的剩余时间内保持该优先级，从而防止任何中间优先级的线程延迟高优先级线程。

尽管我们使用动态技术来处理涉及互斥锁的优先级反转，类型系统仍在两方面发挥作用。首先，为了确保可靠性，类型系统会强制要求互斥锁的优先级确实是其优先级上限，也就是说，使用该互斥锁的每个线程的优先级都低于该互斥锁的优先级。第二项类型系统修改对于正确处理同时使用互斥锁和条件变量的代码是必需的，我们将通过另一个示例说明其必要性。图 6 展示了更贴近实际的生产者-消费者代码实现（这次生产者的优先级高于消费者），其中互斥锁 mut 既用于保护队列操作（这些操作可能不是原子的），也用于确保消费者调用 pop 时队列仍非空（存在多个消费者时这是必要的）。[6] 由于生产者的优先级为高，mut 的优先级必须为高。然而，这意味着消费者的临界区通常以低优先级运行，但如果生产者在该临界区内尝试获取互斥锁，消费者的优先级可能会被动态提升到高。如果发生这种情况，消费者随后可能会在高优先级下调用低优先级条件变量 cv 上的等待操作；这违反了类型安全。

为了防止上述类型安全违规，我们要求每个临界区（即 s 与 $v\ s$ 之间的部分）在其常规优先级和互斥锁 v 的优先级上限下都能通过类型检查，因为临界区可能以其中任一或两种优先级运行。在图 6 中，这意味着消费者代码的类型不正确，因为该临界区无法

[5]我们的语言不允许线程更改自身的优先级，但可以通过生成一个新线程来运行临界区并发出条件变量信号来有效实现这一点，而主线程会等待该条件变量。[6]POSIX 条件变量还要求将 mut 传递给 wait；线程在 cv 上阻塞时，这会释放互斥锁，并在被信号唤醒时重新获取互斥锁。这种行为不会引发安全性问题；事实上，只要 with 块涵盖了所有可能持有锁的位置，线程就可以在 with 块中的任意位置临时释放锁。由于这一问题与类型系统的可靠性无关，为简化起见，我们不将其纳入模型。

带同步的响应式并行

$\rho \in R$ =优先级 $a, b, c \in$ 线程$\alpha \in$ 条件变量 $\beta \in$ 互斥锁 $\eta ::= \alpha \mid \beta$类型 $\tau ::=$ 单位类型 | 自然数 | τ 引用 | 条件变量$[\alpha; \rho]$ | 互斥锁$[\rho]$值 $v ::= x \mid$ 高优先级$\mid \overline{n} \mid$ 条件变量$[\alpha] \mid$ 互斥锁$[\beta] \mid$ 引用$[s]$指令 $i ::= v \mid$ 生成$[\rho]\{s\} \mid$ 引用$[\tau]\ v \mid !v \mid v := v \mid$新建条件变量$[\rho] \mid$ 等待 $v \mid$信号 $v \mid$提升 v至 $\rho \mid$新建互斥锁$[\rho]$语句 $s ::=$ Let $x = i$在 $b \mid$与 $v\ b \mid$如果 v 则 b否则 $b \mid$While $v\ b \mid s; s \mid$跳过

图 7. λ_s^4的语法

由于等待（条件变量），在高优先级下进行类型检查。（通过将条件变量 cv 的优先级改为高优先级，可以使代码通过类型检查。）总之，除了上述针对条件变量的限制外，我们还增加了两项限制：

(6) 优先级为 ρ 的互斥锁的任何临界区，都必须以小于或等于 ρ 的优先级运行（即，ρ 是该互斥锁的优先级上限）。(7) 优先级为 ρ 的互斥锁的任何临界区，都必须在其自身优先级和 ρ 下均通过类型检查。

3 响应性的类型系统

本节通过介绍一个包含线程处理、条件变量和互斥锁的核心演算 λ_s^4，将第 2 节中的思想形式化。

3.1 语法

图 7 展示了 λ_s^4 的语法。优先级 ρ 取自一个全序集 R，其全序关系为 $\preceq$。在程序运行期间，该集合及其全序关系固定不变，但可以任意指定。我们使用元变量 a、b、c 及其变体来表示线程。每个条件变量和互斥锁都有唯一的此类名称，我们将在语义中用它来跟踪获取、发信号等操作。我们使用 α 及其变体作为条件变量的名称，使用 β 表示互斥锁，使用 η 表示条件变量或互斥锁。基本类型包括单位类型和自然数。此外，我们还有类型 τ ref，表示指向类型为 τ 的值的可变引用。条件变量句柄和互斥锁句柄也各有对应的类型。这两种类型都指明句柄的优先级（对于互斥锁而言，即互斥锁的优先级；条件变量则可以有多个优先级不同的句柄）。条件变量句柄的类型还包含与该句柄关联的条件变量名称。

语言的其余部分分为三个层次：值 v、指令 i和语句 s。值包括变量，以及其他不会求值的表达式：单位值；自然数；以及指向条件变量、互斥锁和引用的句柄。指令采用“2/3-CPS”形式：它们的子组件仅包含值，从而简化了语义的呈现。通过 let 绑定中间结果可以构造复杂表达式，因此这种形式不会损失一般性。指令对值执行单个操作并返回一个新值。指令 spawn$[\rho]\{s\}$ 会以优先级 ρ生成一个新线程来运行语句 s。用于操作全局状态的指令包括 ref$[\tau]\ v$，它会创建一个类型为 τ、初始值为 v的新引用；$!v$，它会获取引用 v的值；以及 $v_1 := v_2$，它会将 v_2 赋给引用 v_1。指令 newcv$[\rho]$ 会创建一个新的条件变量，并以优先级 ρ返回其句柄；随后可使用此句柄等待条件变量、发送信号或提升其优先级。

语句处理控制流。语句 Let $x = i$ in s 将 x 绑定到指令 i 的结果，并继续执行 s。否则，语句只包含值和子语句。带有 $v\ s$ 的互斥锁保护的临界区

$$\frac{}{\Gamma, x:\tau \vdash^R_\Sigma x:\tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash^R_\Sigma \langle\rangle : \mathtt{unit}}\ (\mathtt{unitI}) \qquad \frac{}{\Gamma \vdash^R_\Sigma \overline{n} : \mathtt{nat}}\ (\mathtt{natI}) \qquad \frac{}{\Gamma \vdash^R_{\Sigma, s\sim\tau} \mathtt{ref}[s] : \tau\ \mathtt{ref}}\ (\textsc{RefVal})$$

$$\frac{}{\Gamma \vdash^R_{\Sigma,\beta@\rho} \mathtt{mutex}[\beta] : \mathtt{mutex}[\rho]}\ (\textsc{Mutex}) \qquad \frac{}{\Gamma \vdash^R_{\Sigma,\alpha@\rho} \mathtt{cv}[\alpha] : \mathtt{cv}[\alpha;\rho]}\ (\mathrm{CV})$$

图 8. 值类型规则。

$$\frac{\Gamma;\Psi \vdash^R_\Sigma s@\rho' \dashv \Psi'' \qquad \forall\alpha,\rho''.\Psi(\alpha,\rho'') \neq \mathsf{None} \rightarrow \Psi(\alpha,\rho) \neq \mathsf{None}}{\Gamma;\Psi', \Psi \vdash^R_\Sigma \mathtt{spawn}[\rho']\{s\} \leftarrow \mathtt{unit}@\rho \dashv \Psi'}\ (\textsc{Spawn}) \qquad \frac{\Gamma \vdash^R_\Sigma v:\tau}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{ref}[\tau]\ v \leftarrow \tau\ \mathtt{ref}@\rho \dashv \Psi}\ (\textsc{NewRef})$$

$$\frac{\Gamma \vdash^R_\Sigma v : \tau\ \mathtt{ref}}{\Gamma;\Psi \vdash^R_\Sigma !v \leftarrow \tau@\rho \dashv \Psi}\ (\textsc{Deref}) \qquad \frac{\Gamma \vdash^R_\Sigma v_1 : \tau\ \mathtt{ref} \qquad \Gamma \vdash^R_\Sigma v_2 : \tau}{\Gamma;\Psi \vdash^R_\Sigma v_1 := v_2 \leftarrow \mathtt{unit}@\rho \dashv \Psi}\ (\textsc{Update}) \qquad \frac{\Gamma \vdash^R_\Sigma v : \mathtt{cv}[\alpha;\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{wait}\ v \leftarrow \mathtt{unit}@\rho \dashv \Psi}\ (\textsc{Wait})$$

$$\frac{\Gamma \vdash^R_\Sigma v : \mathtt{cv}[\alpha;\rho'] \qquad \Psi(\alpha,\rho) \neq \mathsf{None}}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{signal}\ v \leftarrow \mathtt{unit}@\rho \dashv \Psi}\ (\textsc{Signal})$$

$$\frac{\begin{array}{c}\Gamma \vdash^R_\Sigma v : \mathtt{cv}[\alpha;\rho_1] \\ \Gamma \vdash^R \rho_1 \preceq \rho_2 \qquad \forall\rho_0.(\rho_1 \preceq \rho_0 \wedge \rho_2 \not\preceq \rho_0) \rightarrow \Psi(\alpha,\rho_0) = \mathsf{Owned} \qquad \forall\rho_0.\rho_2 \not\preceq \rho_0 \rightarrow \Psi'(\alpha,\rho_0) = \mathsf{None} \\ \forall\rho_0.\rho_2 \preceq \rho_0 \rightarrow \Psi'(\alpha,\rho_0) = \Psi(\alpha,\rho_0) \qquad \forall\alpha' \neq \alpha,\rho_0.\Psi'(\alpha',\rho_0) = \Psi(\alpha',\rho_0)\end{array}}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{promote}\ v\ \mathtt{to}\ \rho_2 \leftarrow \mathtt{cv}[\alpha;\rho_2]@\rho \dashv \Psi'}\ (\textsc{Promote})$$

$$\frac{\begin{array}{c}\rho' \preceq \rho \qquad \alpha\ \mathtt{fresh} \\ \forall\rho_0,\rho' \preceq \rho_0.\Psi'(\alpha,\rho_0) = \mathsf{Owned} \qquad \forall\rho_0,\rho' \not\preceq \rho_0.\Psi'(\alpha,\rho_0) = \mathsf{None} \qquad \forall\alpha' \neq \alpha.\forall\rho_0.\Psi'(\alpha',\rho_0) = \Psi(\alpha',\rho_0)\end{array}}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{newcv}[\rho'] \leftarrow \mathtt{cv}[\alpha;\rho']@\rho \dashv \Psi'}\ (\textsc{NewCV})$$

$$\frac{}{\Gamma;\Psi \vdash^R_\Sigma \mathtt{newmutex}[\rho'] \leftarrow \mathtt{mutex}[\rho']@\rho \dashv \Psi}\ (\textsc{NewMutex})$$

Fig. 9. Instruction typing rules.

也是一条语句。语句还包括条件语句、While 循环、语句的连接，以及空语句 Skip。

3.2 静态语义

现在我们介绍 λ^4_s 的类型系统。该类型系统结合互斥锁的动态优先级上限机制，可在静态层面确保程序不存在优先级反转。值的类型判断形式为 $\Gamma \vdash^R_\Sigma v\colon \tau$。该判断有两个参数： R 是优先级的全序集。签名 Σ 将引用单元映射到其中所含的类型（这些条目记作 $s{\sim}\tau$），并将互斥锁和条件变量映射到它们的优先级（$\alpha@\rho$ 和 $\beta@\rho$）。对于每个引用单元和互斥锁，签名中至多有一个条目。然而，条件变量可以在签名中出现多次，并对应多个优先级，从而允许同一条件变量的不同句柄具有不同的优先级。上下文 Γ 与通常一样，将变量映射到类型。图 8 中的判断规则相对直接。

指令的类型判断， $\Gamma;\Psi \vdash^R_\Sigma i \sim\colon \tau@\rho \dashv \Psi'$ 更加复杂。它和之前一样使用 R、 Σ和 Γ。与值不同，指令和语句的类型取决于运行该指令时的优先级；此优先级是

带同步的响应式并行

$$\frac{\Gamma; \Psi \vdash^R_\Sigma i \rightsquigarrow \tau@\rho \dashv \Psi' \qquad \Gamma, x : \tau; \Psi' \vdash^R_\Sigma s@\rho \dashv \Psi''}{\Gamma; \Psi \vdash^R_\Sigma \texttt{let}\ x = i\ \texttt{in}\ s@\rho \dashv \Psi''}\ (\textsc{Let}) \qquad \frac{}{\Gamma; \Psi \vdash^R_\Sigma \texttt{skip}@\rho \dashv \Psi}\ (\textsc{Skip})$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{mutex}[\rho'] \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho' \dashv \Psi' \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho \dashv \Psi' \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma; \Psi \vdash^R_\Sigma \texttt{with}\ v\ s@\rho \dashv \Psi'}\ (\textsc{WithLock})$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma; \Psi \vdash^R_\Sigma s_1@\rho \dashv \Psi' \qquad \Gamma; \Psi \vdash^R_\Sigma s_2@\rho \dashv \Psi'}{\Gamma; \Psi \vdash^R_\Sigma \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2@\rho \dashv \Psi'}\ (\textsc{If}) \qquad \frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho \dashv \Psi}{\Gamma; \Psi \vdash^R_\Sigma \texttt{while}\ v\ s@\rho \dashv \Psi}\ (\textsc{While})$$

$$\frac{\Gamma; \Psi \vdash^R_\Sigma s_1@\rho \dashv \Psi' \qquad \Gamma; \Psi' \vdash^R_\Sigma s_2@\rho \dashv \Psi''}{\Gamma; \Psi \vdash^R_\Sigma s_1; s_2@\rho \dashv \Psi''}\ (\textsc{Seq})$$

Fig. 10. Statement typing rules.

$$\begin{array}{ccc} \texttt{Owned} \longrightarrow \texttt{Owned} \otimes \texttt{None} & \texttt{Owned} \longrightarrow \texttt{None} \otimes \texttt{Owned} & \texttt{Owned} \longrightarrow \texttt{Shared} \otimes \texttt{Shared} \\ \texttt{Shared} \longrightarrow \texttt{None} \otimes \texttt{Shared} & \texttt{Shared} \longrightarrow \texttt{Shared} \otimes \texttt{None} & \texttt{Shared} \longrightarrow \texttt{Shared} \otimes \texttt{Shared} \\ & \texttt{None} \longrightarrow \texttt{None} \otimes \texttt{None} & \end{array}$$

图 11. 拆分权限级别的规则。

在判断中表示为 ρ。该判断还会跟踪条件变量的所有权；正如第 2 节所述，条件变量可以处于三种状态之一：无、共享和独占。映射 Ψ 将条件变量名称与优先级的配对映射到“权限”等级，写作 $\Psi(\alpha, \rho) = \pi \in \{$无，共享，独占$\}$。指令可以产生和消耗权限，因此该判断包含指令执行前的映射（Ψ）和执行后的映射（Ψ'）。最后，该判断表示该指令会产生一个类型为 τ的值。

在生成规则中，$\Psi'\Psi$ 表示权限在 Ψ' 和 Ψ之间拆分。为拆分权限级别，我们定义判断 $\pi_1 \rightarrow \pi_2 \otimes \pi_3$，如图 11 所示，该判断表明 π_1 中的权限在 π_2和 π_3之间拆分。形式上，如果 $\Psi'' = \Psi'\Psi$，那么对于所有 α和 ρ，都有 $\Psi''(\alpha, \rho) \rightarrow \Psi'(\alpha, \rho) \otimes \Psi(\alpha, \rho)$。当前线程保留 Ψ'，而 Ψ 则传递给新线程。第二个前提要求，如果 Ψ 在任何优先级上共享或拥有条件变量 α，那么生成线程也必须在其自身优先级上共享或拥有该条件变量。这确保满足第 2 节列表中的限制 3。

NewRef、Deref 和 Update 规则不涉及优先级或权限，而是以标准方式为对引用执行的操作指定类型。Wait 操作要求子表达式具有优先级为 ρ'、指向条件变量 α 的句柄类型。正如前文所述（限制 1），等待要求当前线程的优先级低于该句柄的优先级。等待不要求拥有条件变量的任何所有权。Signal 规则要求当前线程拥有或共享条件变量 α（限制 2）。请注意，发信号还要求线程的优先级大于或等于条件变量的优先级；但要求权限非空已经确保了这一点，因为不存在优先级较低时的任何权限（限制 4）。

Promote 规则会从 CV 句柄的类型中获取 CV 名称和优先级，并检查句柄当前的优先级是否低于 ρ_2，即将提升到的优先级。正如前文的动机说明（限制 5）所述，提升要求在所有不高于 ρ_2 的优先级上都具有 Owned 权限；这一点由第三个前提检查。其余前提定义了 Ψ'，即返回的权限映射。所有不高于 ρ_2 的优先级上针对 α 的权限都会被移除（以满足限制 4），其他权限则保持不变。最后，NewCV 会创建一个新的条件

变量名，并为返回的句柄返回一个新的上下文 Ψ'，其中新条件变量的所有权在相应的优先级上初始化为 Owned 或 None。

语句的判断为 $\Gamma; \Psi \vdash^{R}_{\Sigma} s@\rho \dashv \Psi'$，与指令的判断大体相同，区别在于语句不会返回值。唯一会直接与优先级交互的规则是 WithLock。该规则确保临界区在与线程当前优先级以及互斥锁的优先级上限相对应的情况下都具有良类型（因为临界区在运行时可能会提升到该优先级）：这对应于第 2 节的限制 7。无论优先级如何，临界区都必须能在相同的上下文 Ψ 和 Ψ' 下进行类型检查，这两个上下文会贯穿整个过程。最后一个前提条件强制执行限制 6，即当前线程的优先级必须小于或等于互斥锁的优先级上限。其余规则则将优先级和所有权贯穿各个子语句。Let 规则还会在对语句 s 进行类型检查时，将绑定变量 x 连同其相应类型添加到上下文中。请注意，在 If 语句中，两个分支必须具有相同的输入和输出权限。While 规则要求 while 循环体的权限保持不变；不过，由于我们的权限粒度较粗，这一要求相当宽松。例如，只要共享权限仍为共享，就可以在循环中任意拆分。

4 用于响应式同步的有向无环图模型

本节将建立优先级反转的形式化定义，并以此证明类型系统的正确性。形式化过程如下：我们使用图来构建一种模型，以表示 λ^4_s 程序的并行性和同步。先前的工作 [Muller 等人 2017, 2018, 2020] 已经证明，对于用此类图表示的程序，只要这些图是“良构的”（即不存在优先级反转），就能高效地进行可调度性分析。我们提出了这些技术的一种新推广，并将其应用于 λ^4_s，其中包括对含有互斥锁和条件变量的程序的良构性定义。我们将使用该定义来证明良类型的 λ^4_s 程序不存在优先级反转。我们首先回顾与基于有向无环图的并行性和调度模型相关的预备知识，随后将这些结果扩展至 λ^4_s。

4.1 预备知识：有向无环图模型

我们使用有向无环图（DAG）对并行程序的执行建模，其中顶点表示计算单位（为简化后续概念，我们不失一般性地假设每个顶点表示一个固定的计算时间单位，例如一个处理器周期），边表示程序各部分之间的依赖关系。如果 u 是 u' 在有向无环图中的祖先，记作 $u \sqsupseteq u'$，那么 $u = u'$ 或 u 必须在 u' 之前执行。如果 $u \not\sqsupseteq u'$ 和 $u' \not\sqsupseteq u$，也就是说，图中这两个顶点之间不存在依赖路径，那么 u 和 u' 所表示的计算就可以并行运行。

由于 λ^4_s 允许任意同步，程序可能出现死锁，即两个或多个线程以循环方式相互依赖；这种情况会表现为依赖图中存在一个环。不过，由于很难对存在死锁的程序运行时间得出有意义的结论，我们的研究结果将聚焦于不会发生死锁、其图为无环图的程序。因此，我们将“图”和“有向无环图”这两个术语近似地交替使用。

有向无环图表示并行程序的一种特定执行。对于非确定性程序，同步模式可能取决于许多超出我们推理范围的因素，因此，同一个程序可能产生许多种有向无环图，每种对应一种可能的执行。

有向无环图的记法与术语。我们将使用元变量 g（及其变体）表示图，并将图记作四元组 $(\mathrm{T}\ E^c,\ E^s,\ E^w)$。在这种记法中，T 是一个线程 集合，其余三个分量

则是线程间边的集合。如 λ_s^4 所示，我们使用 a、b、c 及其变体来命名线程。我们用 $a \overset{\rho}{\hookrightarrow} u_1 \cdot u_2 \cdot \ldots \cdot u_n$ 表示一个优先级为 ρ、顶点为 u_1、……、u_n 的单个线程 a。边 (u_1, u_2)、(u_2, u_3)……、(u_{n-1}, u_n) 是隐含的；我们将其称为线程边。我们常用元变量 s 和 t 分别表示一个线程的第一个和最后一个顶点，但其他情况下则使用 u 及其变体表示顶点。集合 E^c、E^s 和 E^w 分别包含创建边、同步边和弱边。创建边的形式为 (u, a)，其中 u 是一个顶点，a 是由计算 u 创建或生成的线程：作为一条边，它可以看作从 u 指向线程 a 的第一个顶点。E^s 和 E^w 中的边形式为 (u_1, u_2)，表示从 u_1 到 u_2 的边。同步边表示线程间的同步。

弱边 [Muller 等人 2020] 用于捕获运行时发生的计算之间的先发生关系，但这些关系并非显式同步的结果。例如，弱边最初用于跟踪由全局内存引起的先发生关系：如果计算 u 将一个值写入内存，而该值随后被计算 $u\prime$ 读取，则两者之间可能存在一条弱边（u，$u\prime$）。如果一条路径（即一系列边）中的所有边都是强边（即不是弱边），则称该路径为强路径。如果 $u \sqsupseteq u\prime$ 与 u 到 $u\prime$ 之间的所有路径都是强路径，则称 u 是 $u\prime$ 的强祖先，记作 $u \sqsupseteq^s u\prime$。如果从 u 到 $u\prime$ 的某条路径包含弱边，则称 u 是 $u\prime$ 的弱祖先，记作 $u \sqsupseteq^w u\prime$。如果路径是弱路径还是强路径并不重要，我们将继续记作 $u \sqsupseteq u\prime$。

调度与响应时间。给定一组处理器和一系列时间步，有向无环图的调度是指在每个时间步将顶点分配给处理器；这相当于在并行机器上执行由有向无环图表示的程序。调度必须遵循有向无环图所表示的依赖关系。如果某个顶点的所有强祖先均已执行，则该顶点就绪；调度只能分配就绪顶点。如果调度还反映了弱边所表示的先发生关系，则称其为可容许调度；在可容许调度中，如果某个顶点已就绪，则其所有弱父节点（即其传入弱边的源）均已执行。

我们希望找到一个有向无环图的调度方案，使特定线程 $a \overset{\rho}{\hookrightarrow} s \cdot$ 响应时间 $\cdot\, t$ 最小。我们将其定义为从 s 变为就绪到 t 执行之间（含首尾）的步数。Brent 定理 [Brent 1974] 及其后续研究成果，为满足特定要求的调度方案提供了近似最优的界限。我们将采用的要求是调度方案必须是 提示式 [Muller 等人 2017, 2018]。在任意时间步，提示式调度方案会先分配所有处于最高优先级的就绪节点，再分配次高优先级的节点，依此类推。只有在没有剩余就绪顶点时，处理器才会空闲。

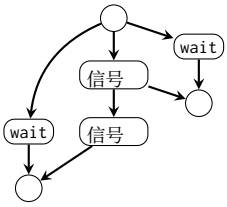
工作量与跨度。调度界通常用两个可从有向无环图中计算的量来表述：工作量 是计算总量，而跨度 则衡量关键路径长度。我们用与这些量密切相关的量来界定响应时间。线程 a（其优先级为 ρ）的竞争者工作量 $W_{\not\prec\rho}(\downarrow a)$，是指优先级不低于 ρ、且可能与 a 的顶点并行执行的顶点数量（工作量）。通俗地说，这些顶点就是线程 a 的顶点可能需要与之“竞争”处理器时间的顶点。竞争者工作量的正式定义如下：

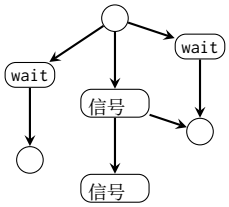
$$W_{\not\prec\rho}(\downarrow a) \triangleq |\{u \in g \mid u \not\sqsupseteq s \wedge t \not\sqsupseteq u \wedge Prio_g(u) \not\prec \rho\}|$$

我们还定义了一个类似于跨度的度量，但该定义不同于以往研究中的定义，因此我们将在本节稍后再介绍。

4.2 互斥锁和条件变量的图模型

现在，我们从较高层面描述如何使用上一小节中的图表示来表示带有互斥锁和条件变量的程序。我们将在第 5 节中对该表示进行形式化定义。由于互斥锁和





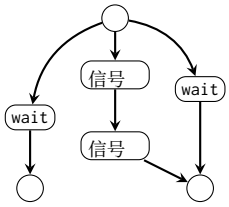


图 12. 展示同一程序中条件变量的几种图。顶点按照执行顺序垂直排列。

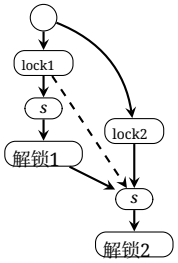


图 13. 两个线程争用一把锁。

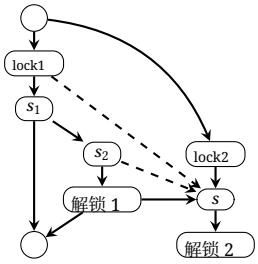


图 14. 线程 1 的优先级被提升至优先级上限。

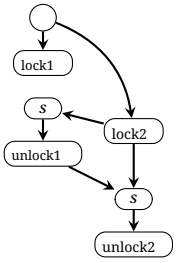


图 15. 图 13 的强化。

条件变量涉及使用一等数据进行同步，而非控制流，因此无法用一个确定的图来表示给定的一段代码。相反，一个图表示程序的一次特定执行，并且必然取决于运行时做出的调度决策。

考虑如何对条件变量上的等待和发信号进行建模，就能清楚地看出这种动态性。假设顶点 u（表示一条指令）向某个条件变量发出信号，从而解除某个线程的阻塞：令 $u_\prime$表示新解除阻塞的线程中的顶点，该顶点因这一信号而变为就绪。我们通过在图中添加同步边 $(u, u_\prime)$，来表示信号操作与等待操作之间的依赖关系。在运行时，很容易确定哪些线程正在某个特定条件变量上阻塞；在形式化操作语义时，我们会这样做。然而，通常无法静态确定这一点，因为在给定的信号操作之前运行了哪些等待操作，取决于相关线程在运行时的具体交错情况，而同一程序的不同运行中，这种情况可能不同。例如，图 12 展示了该程序可能产生的三种有向无环图

[1] 条件变量 cv；生成 {等待（条件变量）(cv)}；生成 {等待（条件变量）(cv)}；信号(cv)；信号(cv)；

含有互斥锁的程序，其图表示同样取决于线程获取互斥锁的顺序。如果顶点 u 释放了互斥锁，并解除对顶点 $u_\prime$的阻塞，就会产生一条同步边 $(u, u_\prime)$。此外，我们从成功获取互斥锁的顶点向 $u_\prime$添加一条弱边，以表示先发生关系：成功获取互斥锁的线程必然先于未成功获取互斥锁的线程运行。图 13 展示了这样一个有向无环图，其中 lock 表示带有 v s 临界区的开始，而 unlock 表示临界区的结束。

该模型还可以表示使用优先级继承或优先级上限等动态机制来防止优先级反转的程序；我们将展示一个优先级上限的图，因为本文重点关注这种机制，但其他机制也可以用类似方式表示。图 14 表示与上述相同的场景，但现在假设获取互斥锁的线程运行优先级低于另外两个线程的优先级以及优先级上限；因此，当 lock2 运行时，第一个线程会提升至该互斥锁的优先级上限。

带同步的响应式并行

我们通过生成另一个线程（运行在优先级上限）来表示这一点，由它完成第一个线程的临界区（图中以 s_2 表示）。从该线程到（仍未成功的）线程还会添加另一条弱边。临界区结束后，第一个线程的控制权会返回到原始线程，并恢复原始优先级；这由从释放（unlock1）操作指向原始线程中下一个操作的同步边表示。

4.3 响应时间与同步

本节的目标是界定程序图中带有互斥锁和条件变量时，在线程 a 的可接受提示调度下的响应时间。下文中，我们用 s 表示 a 的第一个顶点，用 t 表示其最后一个顶点。直观而言，该界限应仅取决于优先级不低于 a的工作量；否则，调度就没有适当地优先处理高优先级工作。上文定义的竞争者工作已仅包含此类工作。因此，我们必须确保，对应于 a关键路径的跨度概念也只包含高优先级计算。为此，我们将讨论范围限制在良构 有向无环图 [Muller 等人2018, 2020], 中，其中低优先级工作不会出现在高优先级线程的关键路径上。这相当于排除优先级反转。

存在弱边时，根据对良构性的直接定义，某些低优先级工作实际上并不位于关键路径上（因为存在先发生关系），却可能看起来 位于关键路径上。例如，在图 13 中，顶点 lock1 看起来位于右侧线程的关键路径上：它是 unlock2（右侧线程的最后一个顶点）的祖先，却不是 lock2（右侧线程的第一个顶点）的祖先。然而，lock2 永远不必等待 lock1 完成（就实际意义而言，这就是“位于关键路径上”的定义），因为弱边表明，lock2 就绪时 lock1 已经完成。为了允许存在此类有向无环图，Muller 等人 [2020] 引入了强化，将带有弱边的图转换为一个图，其中线程 a 关键路径上的所有顶点实际上都表现为 t的强祖先。如果$\hat{g}_a$表示强化后的图，那么所谓的 a-跨度 $S_a(\downarrow u)$ 就是$\hat{g}_a$ 中以 t 结尾的最长强路径，该路径由不是 s祖先的顶点组成。这条路径是强祖先的关键路径：这些祖先在 s 就绪之前可能尚未执行，但必须在 a 完成之前完成。

我们采用类似的方法，但需要注意，同步原语会产生一组不同于以往工作的有向无环图。因此，本文中的有向无环图未必符合该工作中良构性的定义，尽管这些有向无环图仍不包含优先级反转。

我们注意到，要在某个线程的响应时间界中使用其 a跨度，只需图的强化满足两个条件：

- 强化不得移除原图中已有的依赖关系：也就是说，在可容许调度的任意一步中，如果顶点 u 在$\hat{g}_a$中就绪，那么它在 g中也已就绪。
- a跨度（或一般而言，以 t 在$\hat{g}_a$ 中结束且由 s的非祖先组成的任何强路径）不得包含优先级低于 a的工作量。

不同的程序设计模型可能产生不同类型的有向无环图，而这些有向无环图需要不同的强化才能满足上述条件。反过来，原始图必须满足不同的良构条件，才能使强化满足上述两个条件。因此，任何使用弱边的有向无环图模型都可以定义自身的强化概念及相应的良构性概念。只要强化和良构性的定义满足这些条件，该模型中的图就会具有适当的响应时间界。

上文。定理4.1 将这一一般性结果形式化。该界限背后的直觉是，在每个时间步，根据提示调度的定义，我们要么完成 P 单位类型的竞争者工作，要么（如果没有足够的高优先级工作准备就绪）将 a-跨度的长度缩短 1。

定理 4.1。 设 g 是一个图，$\hat{g}_a$ 是其针对线程 $a \overset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in g$ 的 a-强化。假设 g 和 $\hat{g}_a$ 满足以下两个性质。

(1) 如果在可容许调度的任意一步中，顶点 u 在 $\hat{g}_a$ 中已就绪，那么它在 g 中也已就绪。
(2) 如果在 $\hat{g}_a$ 中存在一条从 u 到 t 的强路径，且 $u \not\sqsupseteq s$ 位于 $\hat{g}_a$ 中，那么 $\rho \preceq \mathit{Prio}_g(u)$。

那么，对于 g 在 P 个核上的任意可容许提示调度，

$$T(a) \leq \frac{1}{P}\left[W_{\not\prec\rho}(\downarrow\!\!\!\!\uparrow a) + (P-1)S_a(\downarrow\!\!\!\!\uparrow a)\right]$$

证明。该证明实际上与 [Muller 等人 2020], 的定理 2.1 的证明完全相同；该定理仅假设图及其强化满足这两个性质。□

现在，我们给出本文将使用的良构性（定义 1）和强化（定义 2）的定义，并证明它们满足上述条件，从而保证该界成立。非正式地说，除了对优先级施加预期的限制外，我们还要求：任何因线程 a 等待线程 b 持有的锁而产生的同步边之前，都必须有一条弱边，表明线程 b 在线程 a 尝试获取该锁之前已获取该锁。

定义 1. 如果对于所有线程 $a \overset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in \mathrm{T}$，DAG $g = (\mathrm{T}, E^c, E^s, E^w)$ 都是良构的，

(1) 如果 $u \sqsupseteq^s t$ 和 $u \not\sqsupseteq s$，则 $\rho \preceq \mathit{Prio}_g(u)$。(2) 如果 $u \sqsupseteq^s t$ 和 $u \not\sqsupseteq s$，并且存在一条强边 (u', u)，其中 $u' \sqsupseteq^w t$，那么存在一条弱边 (u, u'')，其中 $u'' \sqsupseteq^s t$，且 $u'' \not\sqsupseteq s$，并且 u'' 不是其线程中的第一个顶点。此外，从 u'' 的父节点出发的边，是指向 u'' 的唯一强边。

由于良构性保证存在弱边，线程 b 的祖先不在线程 a 的关键路径上。强化通过移除线程 b 中顶点 u 的祖先，明确体现了这一点。为避免使其断开，我们将 u 设为线程 a 中紧邻尝试获取锁的顶点之前那个顶点的强子节点；我们将这个顶点称为 u'', p。这样做不会增大响应时间界；根据弱边， u 的父节点必须先于 u'', p 的子节点执行。从响应时间界的角度来看，最坏的情况是 u 直到 u'', p 执行后才就绪，而新增的边明确体现了这一点。例如，图 15 展示了对图 13 中图的强化。请注意，lock1 不再出现在第二个线程的关键路径上，但调度保持不变。

定义 2. Let g 是一个良构图，其线程为 $a \overset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t$。我们将其 a-强化定义如下。对于每条强边 (u, u)，其中 $u' \sqsupseteq^w t$ 且 $u \sqsupseteq^s t$ 和 $u \not\sqsupseteq s$:

(1) 移除根据定义 1 存在的弱边（u, u''）。（2）添加一条强边（u'', p, u），其中 u'' p 是 u'' 的父节点，该强边根据定义 1 存在。（3）移除强边（u, u）。

带同步的响应式并行

$$\begin{array}{llll} \textit{Frames} & f & ::= & -; s \mid \texttt{with*}\ v\ - \mid \texttt{with*}[a; \rho]\ v\ - \mid \texttt{let}\ x = -\ \texttt{in}\ b \\ \textit{Stacks} & k & ::= & \epsilon \mid k :: f \\ \textit{States} & K & ::= & k \triangleright i \mid k \triangleleft v \mid k \blacktriangleright s \mid k \blacktriangleleft \end{array}$$

图 16. 栈语法

定理 4.2。 *Let* g 是一个依据定义 1 的良构图，并令 $\overset{a \hookrightarrow s}{\rho}$ ·。。。· $t \in g$。对于在 P 个核上对 g 进行的任意可容许提示调度，

$$T(a) \leq \frac{1}{P}\left[W_{\not\prec \rho}(\downarrow\hat{\uparrow} a) + (P - 1)S_a(\downarrow\hat{\uparrow} a)\right]$$

证明。我们只需证明定理 4.1 的两个条件。

(1) 设 u 在 $\hat{g}_a$ 中已就绪。根据定义， u 的所有祖先都已执行。我们必须证明 u 在 g 中的所有强祖先都已执行。除非 u 是定义 2 中的 u'' 或 u，否则 u 的祖先在 $\hat{g}_a$ 和 g 之间保持不变。如果 u 是该定义中的 u''，那么 g 中只多了一条弱边，它不会妨碍 u 保持就绪。如果 u 是该定义中的 u，那么根据构造， u 在 $\hat{g}_a$ 中唯一的强父节点是 u''^p，因此该顶点已执行。 u 在 g 中的父节点是定义 2 中的 u'。但是，由于存在一条从 u' 指向 u'' 的弱边，且 u'' 唯一的强父节点已执行， u' 在可容许调度中也必定已执行。因此， u 在 g 中已就绪。

(2) Let u，使得在 $\hat{g}_a$ 和 $u \not\sqsupseteq s$ 中存在一条从 u 到 t 的强路径。我们注意到 $u \sqsupseteq^s t$ 属于 $\hat{g}_a$，因为如果 $u \sqsupseteq^w t$，那么就存在一条强边 (u, u') 和一条从 u' 到 t 的强路径，因此边 (u, u') 本应已从 $\hat{g}_a$ 中移除，这就产生了矛盾。此外，如果 $u \sqsupseteq^s t$ 属于 $\hat{g}_a$，那么 $\rho \leq \mathrm{Prio}_g(u)$。为此，只需证明：若 $u \sqsupseteq^s t$ 属于 $\hat{g}_a$，则 $u \sqsupseteq^s t$ 属于 g。考虑 $\hat{g}_a$ 中一条从 u 到 t 的路径，并考察该路径中一条不属于 g 的边 $(u_1, u_2$。这必定是定义 2 中的边 (u''^p, u)。但根据定义 1，我们有 $u'' \sqsupseteq^s t$ 属于 g，因此 $u''^p \sqsupseteq^s t$ 属于 g。

□

5 优先级反转类型系统的正确性

在本节中，我们证明第3节的类型系统保证不会发生优先级反转。具体而言，我们将证明，与良类型 λ_s^4 程序对应的图根据定义 1 是良构的，因此可以由定理 4.2 进行响应式调度。我们首先给出一个代价语义，用于 λ_s^4 求值程序并生成与执行对应的图。接下来，我们给出主要结果。最后，在第 5.3 节中，我们简要讨论 λ_s^4 的若干扩展，这些扩展对其他常见的同步用例进行建模。

5.1 代价语义

λ_s^4 的动态语义在求值程序的同时，生成一个表示并行执行的图。对于非确定性程序，该图会反映所执行程序运行过程中作出的“选择”，同时也会指出哪些位置存在并行性，因此允许采用与相同“选择”一致的其他调度。构造一种能够完整模拟程序提示和可容许调度的并行操作语义，超出了本文的讨论范围；这种语义将遵循先前工作的思路 [Muller 等人 2020]。

动态语义通过控制栈来定义。栈的语法如图 16 所示。栈由帧组成，每个帧都是带有一个“洞”–的语句，用来表示程序的续延。由于

$$\frac{\rho \in R'}{\vdash^R_\Sigma \epsilon \blacktriangleleft : (\Psi, R') \rightsquigarrow (\Psi, \rho)}\ \text{(KS-Empty)} \qquad \frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_2, R_1) \rightsquigarrow (\Psi_3, \rho_2) \qquad \forall \rho \in R_1 . \cdot; \Psi_1 \vdash^R_\Sigma s@\rho \dashv \Psi_2}{\vdash^R_\Sigma k :: -; s \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_3, \rho_2)}\ \text{(KS-Seq)}$$

$$\frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho_2) \qquad \cdot \vdash^R_\Sigma v : \texttt{mutex}[\rho']}{\vdash^R_\Sigma k :: \texttt{with*}\ v - \blacktriangleleft : (\Psi_1, R_1 \cup \{\rho'\}) \rightsquigarrow (\Psi_2, \rho_2)}\ \text{(KS-WithLockS)}$$

$$\frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho) \qquad \cdot \vdash^R_\Sigma v : \texttt{mutex}[\rho']}{\vdash^R_\Sigma k :: \texttt{with*}[a; \rho]\ v - \blacktriangleleft : (\Psi_1, R_1 \cup \{\rho'\}) \rightsquigarrow (\Psi_2, \rho')}\ \text{(KS-WithLockSP)}$$

$$\frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_2, R_1) \rightsquigarrow (\Psi_3, \rho_2) \qquad \forall \rho \in R_1 . x : \tau; \Psi_1 \vdash^R_\Sigma s@\rho \dashv \Psi_2}{\vdash^R_\Sigma k :: \texttt{let}\ x = -\ \texttt{in}\ s \triangleleft : (\tau, \Psi_1, R_1) \rightsquigarrow (\Psi_3, \rho_2)}\ \text{(KS-Let)}$$

$$\frac{\vdash^R_\Sigma k \triangleleft : (\tau, \Psi_2, R_1) \rightsquigarrow (\Psi_2, \rho_3) \qquad \forall \rho \in R_1 . \cdot; \Psi_1 \vdash^R_\Sigma i \div \tau@\rho \dashv \Psi_2}{\Psi_1 \vdash^R_\Sigma k \triangleright i \mathbin{@} \rho_2 \dashv \Psi_3}\ \text{(KS-PopInstr)}$$

$$\frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_2, R_1) \rightsquigarrow (\Psi_3, \rho_2) \qquad \forall \rho \in R_1 . \cdot; \Psi_1 \vdash^R_\Sigma s@\rho \dashv \Psi_2}{\Psi_1 \vdash^R_\Sigma k \blacktriangleright s \mathbin{@} \rho_2 \dashv \Psi_3}\ \text{(KS-PopStmt)}$$

$$\frac{\vdash^R_\Sigma k \triangleleft : (\tau, \Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho_2) \qquad \cdot \vdash^R_\Sigma v : \tau}{\Psi_1 \vdash^R_\Sigma k \triangleleft v \mathbin{@} \rho_2 \dashv \Psi_2}\ \text{(KS-PushInstr)} \qquad \frac{\vdash^R_\Sigma k \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho_2)}{\Psi_1 \vdash^R_\Sigma k \blacktriangleleft \mathbin{@} \rho_2 \dashv \Psi_2}\ \text{(KS-PushStmt)}$$

图 17. 栈和栈状态的类型规则。

指令采用“2/3-cps”形式，帧只由语句组成，不包含指令或表达式。两个帧 with* v – 和 with*[a; ρ] v –表示运行时出现的临界区状态，稍后会对此进行说明。空栈记为 ϵ。线程当前的执行状态由栈状态 K表示；每个状态都包含表示当前续延的栈。状态 $k \triangleright i$表示线程正在执行指令 i； $k \triangleleft v$表示一条指令刚刚返回值 v； $k \blacktriangleright s$表示线程正在执行语句 s；而 $k \blacktriangleleft$ 表示一条语句刚刚返回（语句不会返回值，因此无需返回值）。

判断 $\vdash^R_\Sigma k \blacktriangleleft : (\Psi, R_1) \rightsquigarrow (\Psi', \rho_2)$ 表示， k需要权限 Ψ 才能继续，并会留下 Ψ'。它还要求当前语句在集合 R_1 中的所有优先级下均为良类型，并表示整个栈可以在 ρ_2 下运行。 $\rho_2 \in R_1$ 是一个不变式。判断 $\vdash^R_\Sigma k \triangleleft : (\tau, \Psi, R_1) \rightsquigarrow (\Psi', \rho_2)$ 与之类似，但适用于一种栈：它正等待指令返回类型为 τ 的返回值。这些判断的规则显示在图 17 的上半部分。栈状态使用判断 $\Psi \vdash^R_\Sigma K \mathbin{@} \rho \dashv \Psi'$ 进行类型检查，该判断表示 K 在优先级 ρ 下、以起始权限 Ψ 为条件时是良类型，并会留下权限 Ψ'。图 17 下半部分给出的这一判断规则，只需将栈的要求（由栈类型判断表示）与语句、指令或返回值相匹配。

动态语义作用于线程池，即由就绪线程组成的集合。一个线程 a 以优先级 ρ运行栈状态 K，其签名为 Σ，记作 $a \overset{K}{\hookrightarrow}_{\rho, \Sigma}$。线程池通过运算符 $\otimes$组合。一个配置 $\mu; \sigma; W; L$ 表示运行中程序的完整快照。它包含一个线程池 μ以及内存 σ，内存是从引用单元（记作 s）到其中所含值的映射。配置还包含两个额外的组件： W 是从互斥锁和条件变量到正在等待该互斥锁的线程的映射。

带同步的响应式并行

或条件变量，并且 L 是从互斥锁映射到当前持有该互斥锁的线程，或 $\perp$ 表示该互斥锁已解锁。对于互斥锁 β，$W(\beta)$ 的一个元素的形式为$(a,(u_1,u_2)\ \rho,\Sigma,K)$，表示一个线程 a处于优先级 ρ、状态为 K 且签名为 ρ。两个顶点 u_1 和 u_2 分别表示互斥锁获取以及互斥锁获取之后的顶点；稍后将使用这些顶点向图中添加适当的同步边和弱边。对于CV α，$W(\alpha)$ 的一个元素与此类似：在顶点对(u_1,u_2) 中，u_1 表示等待操作，而 u_2表示后续顶点。$L(\beta)$ 的一个元素形式为(a,u_1,u_2)，表示一个线程 a持有该锁；如上所述，u_1 和 u_2 分别表示获取操作和后续顶点。

我们为配置的类型检查提出另一条类型规则：

$$\frac{\begin{array}{c}\bigcup_{\eta\in Dom(W)} W(\eta)=\{(a_{n+1},u_{n+1},\rho_{n+1},\Sigma_{n+1},s_{n+1}),\ldots,(a_m,u_m,\rho_m,\Sigma_m,s_m)\}\\ \alpha@\rho\in\Sigma_1,\ldots,\Sigma_n\to\forall i\in[1,m].\forall\rho'\prec\rho.\Psi_i(\alpha,\rho')=\mathsf{None}\qquad\forall i\in[1,m].\Psi_i\vdash^R_{\Sigma,\Sigma_i}K_i\ @\ \rho_i\dashv\Psi_i'\\ \mathtt{Splittable}(\Psi_0,\ldots,\Psi_m)\qquad\mathtt{Splittable}(\Psi_0',\ldots,\Psi_m')\qquad\leq 1\qquad\forall i\in[1,m].\Sigma_i\vdash\sigma\end{array}}{\Sigma\vdash^R a_1\xhookrightarrow[\rho_1,\Sigma_1]{}K_1\otimes\ldots\otimes a_n\xhookrightarrow[\rho_n,\Sigma_n]{}K_n;\sigma;W;L}\ (\textsc{Global})$$

该规则要求线程池中的每个线程（以及阻塞在互斥锁或条件变量上的线程，这些线程不包含在线程池中）都应当在其优先级以及适当的权限映射下，依据其签名是良类型的。它还要求权限是“可拆分的”：我们定义 Splittable$(\Psi_0,\ldots,\Psi_m)$，其含义是：对于所有 α以及所有 ρ，如果 $\Psi_i(\alpha,\rho)=$独占，那么对于所有 $j\neq i$，都有 $\Psi_j(\alpha,\rho)=$None。最后，内存必须相对于每个线程的签名都是良类型的。这由判断 $\Sigma\vdash\sigma$ 表示，该判断要求对于所有 $s{\sim}\tau\in\Sigma$，都有 $\sigma(s)=v$并且 $\cdot\vdash^R_\Sigma v:\tau$。

动态语义判断为 $\mu;\sigma;W;L\mid g\Rightarrow\mu\prime;\ \sigma\prime;W\prime;L\prime\mid g\prime$，表示配置执行一步；如果在执行这一步之前 DAG 为 g，那么之后它就是 $g\prime$。规则见图18–20。大多数规则都会创建一个新顶点 u并将其添加到图中：记号 $g\oplus_a(u,\Sigma)$会将顶点 u添加到 g中的线程 a上。我们还将线程在程序中该位置的签名与每个顶点关联起来。我们还将使用记号 $g\cup E$把 E中的边添加到图中：我们分别将创建边、同步边和弱边记为$(u_1,u_2)^c$、$(u_1,u_2)^s$ 和$(u_1,u_2)^w$。

最不寻常的规则是处理条件变量和互斥锁的规则。等待规则会将当前线程添加到 W。请注意，使用“2/3-cps”形式意味着我们不必将子表达式求值为条件变量句柄；语法和类型规则保证它只能采用这种形式。等待规则会向线程添加两个顶点；一个表示等待操作本身，另一个表示线程中等待操作之后的下一项操作；信号将启用这项操作，因此会将该顶点添加到 W，以便稍后添加一条指向它的同步边。信号有两条规则，具体取决于当前是否有线程在等待条件变量。如果有（Signal1），则选取其中一个；将其从 $W(\alpha)$中移除，并添加回线程池。我们还会向有向无环图添加同步边（u，$u\prime$），其中 u是表示信号的顶点，而 $u\prime$是等待线程中等待操作之后的顶点。如果没有线程在等待（Signal2），则将顶点 u 添加到有向无环图中，但不执行其他操作。

获取锁以开始 with v s 临界区有四条规则。如果锁当前未上锁，WithLockS1 会通过将 $L(\beta)$更新为当前线程，将其转换为上锁状态。与 Wait 一样，我们会向线程添加两个顶点，分别表示获取操作和临界区中的第一个操作。同时，语句也会改为 with* 互斥锁$\lfloor\beta\rceil$ s，以表示该锁已被获取。如果锁当前由优先级高于当前线程的线程持有（WithLockS2），则会将当前线程添加到 $W(\beta)$，并添加一条弱边

$$\frac{u, b \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{spawn}[\rho']\{s'\} \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleleft \langle\rangle \otimes b \underset{\rho,\Sigma}{\hookrightarrow} \epsilon \triangleright s'\mu; \sigma; W; L \mid g \oplus_a (u, \Sigma) \cup \{(u, b)^c\}} \text{ (Spawn)}$$

$$\frac{u, s \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{ref}[\tau]\ y s \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma,s\sim\tau}{\hookrightarrow} k \triangleleft \texttt{ref}[s] \otimes \mu; \sigma[s \mapsto (y, u, \Sigma)]; W; L \mid g \oplus_a (u, \Sigma)} \text{ (NewRef)}$$

$$\frac{u \text{ fresh} \qquad \sigma(s) = (v, u', \Sigma')}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright !\texttt{ref}[s] \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma,\Sigma'}{\hookrightarrow} k \triangleleft v \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)\}} \text{ (Deref)}$$

$$\frac{u \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{ref}[s] := y \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleleft \langle\rangle \otimes \mu; \sigma[s \mapsto (y, u, \Sigma)]; W; L \mid g \oplus_a (u, \Sigma)} \text{ (Update)}$$

$$\frac{u, \alpha \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{newcv}[\rho'] \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma,\alpha@\rho'}{\hookrightarrow} k \triangleleft \texttt{cv}[\alpha] \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{ (NewCV)}$$

$$\frac{u_1, u_2 \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{wait cv}[\alpha] \otimes \mu; \sigma; W; L \mid g \Rightarrow \mu; \sigma; W[\alpha \mapsto (a, (u_1, u_2), \rho, \Sigma, k \triangleleft \langle\rangle) :: W(\alpha)]; L \mid g \oplus_a (u, \Sigma)} \text{ (Wait)}$$

$$\frac{u \text{ fresh} \qquad W(\alpha) = (b, (\_, u'), \rho_b, \Sigma_b, K_b) :: \ell}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{signal cv}[\alpha] \otimes \mu; \sigma; W; L \mid g \qquad \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleleft \langle\rangle \otimes b \underset{\rho_b,\Sigma_b}{\hookrightarrow} K_b \otimes \mu; \sigma; W[\alpha \mapsto \ell]; L \mid g \oplus_a (u, \Sigma) \cup \{(u, u')^s\}} \text{ (Signal1)}$$

$$\frac{u \text{ fresh} \qquad W(\alpha) = [\,]}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{signal cv}[\alpha] \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleleft \langle\rangle \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{ (Signal2)}$$

$$\frac{u \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{promote cv}[\alpha] \texttt{ to } \rho' \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma,\alpha@\rho'}{\hookrightarrow} k \triangleleft \texttt{cv}[\alpha] \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{ (Promote)}$$

$$\frac{u, \beta \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \triangleright \texttt{newmutex}[\rho'] \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma,\beta@\rho'}{\hookrightarrow} k \triangleleft \texttt{mutex}[\beta] \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{ (NewMutex)}$$

$$\frac{}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \texttt{let } x = i \texttt{ in } s \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k :: \texttt{let } x = - \texttt{ in } s \triangleright i \otimes \mu; \sigma; W; L \mid g} \text{ (Let1)}$$

$$\frac{}{a \underset{\rho,\Sigma}{\hookrightarrow} k :: \texttt{let } x = - \texttt{ in } s \triangleleft v \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright [v/x]s \otimes \mu; \sigma; W; L \mid g} \text{ (Let2)}$$

图 18. 动态语义规则（第 1 部分）

从 $u_{\prime}1$（即成功获取锁的顶点）指向 u_2（当前线程中新添加的顶点）。如果锁当前由优先级不高于当前线程的线程持有，则将持有锁的线程提升至优先级 $\rho\beta$，即互斥锁的优先级上限。如果持有锁的线程处于活动状态，则由 WithLockS3 执行此操作；如果该线程正在等待其他锁或条件变量，则由 WithLockS4 执行。该规则会在$\rho\beta$处创建一个新线程 $b_{\prime}$，以运行临界区的剩余部分。辅助判断 $K_b \underset{\beta,b,\rho'}{\rightsquigarrow} K_{\prime\prime}$ 会将临界区替换为形如 with*$[b; \rho_{\prime}]\ v\ s$的形式，表示临界区结束时线程应将控制权返回给线程 b，位置为 $\rho_{\prime}$。此判断的规则定义于图 21 中。随后会添加两个新顶点 $u_{\prime}1$ 和 $u_{\prime\prime}2$，以启动线程 $b_{\prime}$——需要通过更新 $L(\beta)$ 来表明 $b_{\prime}$ 现在持有该锁。最后，从 $u_{\prime}1$ 向 $W(\beta)$中的每个其他线程添加弱边。

带同步的响应式并行

$$\frac{u_1,\ u_2 \text{ 新鲜} \qquad L(\beta) = \bot}{a \xhookrightarrow[\rho,\Sigma]{} k \blacktriangleright \text{使用互斥锁}[\beta]\ s \otimes \mu;\ \sigma;\ W;\ L \mid g \Rightarrow a \xhookrightarrow[\rho,\ \Sigma]{} k :: \text{使用* 互斥锁}[\beta] - \blacktriangleright s \otimes \mu;\ \sigma;\ W;\ L[\beta \mapsto (a,\ u_1,\ u_2)\ ] \mid g \oplus_a (u,\ \Sigma)} \text{(WithLockS1)}$$

$$\frac{u_1,\ u_2 \text{ 新鲜} \qquad L(\beta) = (a', u_1', u_2') \qquad \rho \preceq PRIO_g(a') \qquad K' = k :: \text{使用* 互斥锁}[\beta] - \blacktriangleright s}{a \xhookrightarrow[\rho,\Sigma]{} k \blacktriangleright \text{使用互斥锁}[\beta]\ s \otimes \mu; \sigma; W; L \mid g \Rightarrow \mu; \sigma; W[\beta \mapsto (a,\ (u_1,\ u_2)\ \rho,\ \Sigma,\ K') :: W(\beta)\ ]; L \mid g \oplus_a (u_1,\ \Sigma)\ \oplus_a (u_2,\ \Sigma)\ \cup \{(u_1',\ u_2)^w\}} \text{(WithLockS2)}$$

$$\frac{K' = k :: \text{使用互斥锁}[\beta] - \blacktriangleright s \qquad \begin{array}{c} u_1,\ u_2,\ u_1'',\ u_2'' \text{ 新鲜} \qquad L(\beta) = (b, u_1', u_2') \qquad \rho \not\preceq \rho' \\ \beta @ \rho_\beta \in \Sigma \qquad E^w = \{(u_1'',\ u_{2c})^w \mid (c,\ (u_{1c},\ u_{2c})\ \rho_c,\ \Sigma_c,\ s_c) \in W(\beta)\} \end{array} \qquad K_b \underset{\beta, b, \rho'}{\rightsquigarrow} K''}{\begin{array}{l} a \xhookrightarrow[\rho,\Sigma]{} k \blacktriangleright \text{使用互斥锁}[\beta]\ s \otimes b \xhookrightarrow[\rho',\Sigma']{} K_b \otimes \mu; \sigma; W; L \mid g \\ \Rightarrow b' \xhookrightarrow[\rho_\beta]{} K'' \otimes \mu;\ \sigma;\ W[\beta \mapsto (a,\ (u_1,\ u_2)\ \rho,\ \Sigma,\ K') :: W(\beta)\ ];\ L[\beta \mapsto (b',\ u_1'',\ u_2'')\ ] \mid \\ g \oplus_{b'} (u_1'', \Sigma')\ \oplus_{b'} (u_2'', \Sigma')\ \oplus_a (u_1, \Sigma)\ \oplus_a (u_2, \Sigma)\ \cup \{(u_1'', u_2)^w, (\text{最后一个 } Vg(a')\ u_1'')^c\} \end{array}} \text{(WithLockS3)}$$

$$\frac{\begin{array}{c} u_1, u_2, u_1'', u_2'' \text{ 新鲜} \\ L(\beta) = (b,\ u_1',\ u_2') \qquad W(\eta') = \ell_1 :: (b, \_, \rho', \Sigma', K_b) :: \ell_2 \qquad \rho \not\preceq \rho' \qquad K' = k :: \text{使用互斥锁}[\beta] - \blacktriangleright s \qquad \beta @ \rho_\beta \in \Sigma \\ E^w = \{(u_1'',\ u_{2c})^w \mid (c,\ (u_{1c},\ u_{2c})\ \rho_c,\ \Sigma_c,\ s_c) \in W(\beta)\} \qquad K_b \underset{\beta,\ b,\ \rho'}{\rightsquigarrow} K'' \qquad W'(\eta') = \ell_1 :: (b',\ \_,\ \rho_\beta,\ \Sigma',\ K'') :: \ell_2 \end{array}}{\begin{array}{l} a \xhookrightarrow[\rho,\ \Sigma]{} k \blacktriangleright \text{使用互斥锁}[\beta]\ s \otimes \mu;\ \sigma;\ W;\ L \mid g \Rightarrow \mu;\ \sigma;\ W[\beta \mapsto (a,\ (u_1,\ u_2)\ \rho,\ \Sigma,\ K') :: W(\beta)\ ];\ L[\beta \mapsto (b',\ u_1'',\ u_2'')\ ] \mid \\ g \oplus_{b'} (u_1'',\ \Sigma')\ \oplus_{b'} (u_2'',\ \Sigma')\ \oplus_a (u_1,\ \Sigma)\ \oplus_a (u_2,\ \Sigma)\ \cup \{(u_1'',\ u_2)^w,\ (LastVg(a')\ u_1'')^c\} \end{array}} \text{(WithLockS4)}$$

$$\frac{u \text{ 新鲜} \qquad W(\beta) = []}{a \xhookrightarrow[\rho,\ \Sigma]{} k :: \text{带锁}[\beta] - \blacktriangleleft \otimes \mu; \sigma; W; \quad L \mid g \Rightarrow a \xhookrightarrow[\rho,\ \Sigma]{} k \blacktriangleleft \otimes \mu; \sigma; W; L[\beta \mapsto \bot] \mid g \oplus_a (u,\ \Sigma)} \text{(WithLockE1)}$$

$$\frac{\begin{array}{c} u \text{ 新鲜} \qquad W(\beta) = \ell_1, (b, (u_1, u_2)\ \rho_b, \Sigma_b, K_b) :: \ell_2 \\ \forall (\_, \_, \rho_c, \_, \_) \in \ell_1 \ell_2, \rho_c \preceq \rho_b \qquad E^w = \{(u_1, u_2')^w \mid (c, (u_1', u_2')\ \rho_c, \Sigma_c, s_c) \in \ell_1 \ell_2\} \end{array}}{\begin{array}{l} a \xhookrightarrow[\rho,\ \Sigma]{} k :: \text{使用* 互斥锁}[\beta] - \blacktriangleleft \otimes \mu; \sigma; W; L \mid g \\ \Rightarrow a \xhookrightarrow[\rho,\Sigma]{} k \blacktriangleleft \otimes b \xhookrightarrow[\rho_b, \Sigma_b]{} K_b \otimes \mu; \sigma; W[\beta \mapsto \ell_1 \ell_2]; L[\beta \mapsto (b, u_1, u_2)\ ] \mid g \oplus_a (u, \Sigma)\ \cup \{(u, u_2)^s\}\ \cup E^w \end{array}} \text{(WithLockE2)}$$

$$\frac{u,\ u' \text{ 新鲜} \qquad W(\beta) = []}{a \xhookrightarrow[\rho,\Sigma]{} k :: \text{with*}[a';\ \rho'] \text{互斥锁}[\beta] - \blacktriangleleft \otimes \mu;\ \sigma;\ W;\ L \mid g \Rightarrow a' \xhookrightarrow[\rho'\ \Sigma]{} k \blacktriangleleft \otimes \mu;\ \sigma;\ W;\ L[\beta \mapsto \bot] \mid g \oplus_a (u,\ \Sigma)\ \oplus_{a'} (u',\ \Sigma)\ \cup \{(u,\ u')^s\}} \text{(WithLockE3)}$$

$$\frac{\begin{array}{c} u,\ u' \text{ 新鲜} \qquad W(\beta) = \ell_1,\ (b,\ (u_1,\ u_2)\ \rho_b,\ \Sigma_b,\ K_b) :: \ell_2 \\ \forall (\_, \_, \rho_c, \_, \_) \in \ell_1 \ell_2, \rho_c \preceq \rho_b \qquad E^w = \{(u_1,\ u_2')^w \mid (c,\ (u_1',\ u_2')\ \rho_c,\ \Sigma_c,\ K_c) \in \ell_1 \ell_2\} \end{array}}{\begin{array}{l} a \xhookrightarrow[\rho,\ \Sigma]{} k :: \text{with*}[a';\ \rho'] \text{互斥锁}[\beta] - \blacktriangleleft \otimes \mu; \sigma; W; L \mid g \\ \Rightarrow a' \xhookrightarrow[\rho',\Sigma]{} k \blacktriangleleft \otimes b \xhookrightarrow[\rho_b \quad ,\Sigma_b]{} K_b \otimes \mu; \sigma; W[\beta \mapsto \ell_1 \ell_2]; L[\beta \mapsto (b, u_1, u_2)\ ] \mid \\ g \oplus_a (u,\ \Sigma)\ \oplus_{a'} (u',\ \Sigma)\ \cup \{(u,\ u_2)^s,\ (u,\ u')^s\}\ \cup E^w \end{array}} \text{(WithLockE4)}$$

图 19. 动态语义规则（第 2 部分）

结束临界区也有四条规则。如果线程尚未被提升到更高的优先级，且没有其他线程正在等待锁，规则 WithLockE1 会简单地将 $L(\beta)$更新为 $\bot$，从而释放锁。如果有其他线程正在等待，则从 $W(\beta)$中选择优先级最高的线程 b。规则WithLockE2将 $L(\beta)$赋值给 b，并更新等待线程列表。该规则还会从表示锁释放的顶点向 b 中获取锁的顶点添加一条同步边。最后，该规则会向等待线程添加弱边。如果持有锁的线程已被提升到锁的优先级上限，则相应的规则（如果没有其他线程正在等待，则为 WithLockE3，否则为 WithLockE4）还会将该线程移回其原始名称和优先级。

$$\frac{n > 0 \qquad u \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \texttt{if } \overline{n} \texttt{ then } s_1 \texttt{ else } s_2 \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright s_1 \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{(If1)}$$

$$\frac{n = 0 \qquad u \text{ fresh}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \texttt{if } \overline{n} \texttt{ then } s_1 \texttt{ else } s_2 \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright s_2 \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{(If2)}$$

$$\frac{}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \text{当 } v\ s \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \text{如果 } v \text{则 } (s; \text{当 } v\ s) \text{否则跳过 } \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{(While)}$$

$$\frac{u \text{ 新鲜}}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright \texttt{Skip} \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleleft \otimes \mu; \sigma; W; L \mid g \oplus_a (u, \Sigma)} \text{(Skip)}$$

$$\frac{}{a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright s_1; s_2 \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k :: -; s_2 \blacktriangleright s_1 \otimes \mu; \sigma; W; L \mid g} \text{(Seq1)}$$

$$\frac{}{a \underset{\rho,\Sigma}{\hookrightarrow} k :: -; s_2 \blacktriangleleft \otimes \mu; \sigma; W; L \mid g \Rightarrow a \underset{\rho,\Sigma}{\hookrightarrow} k \blacktriangleright s_2 \otimes \mu; \sigma; W; L \mid g} \text{(Seq2)}$$

图 20. 动态语义规则（第3部分）

$$\frac{}{k :: \texttt{with* mutex}[\beta] - \underset{\beta,a,\rho}{\rightsquigarrow} k :: \texttt{with*}[a; \rho]\ \texttt{mutex}[\beta] -} \qquad \frac{k \underset{\beta,a,\rho}{\rightsquigarrow} k'}{f \neq \texttt{with* mutex}[\beta] - \quad k :: f \underset{\beta,a,\rho}{\rightsquigarrow} k' :: f} \qquad \frac{k \underset{\beta,a,\rho}{\rightsquigarrow} k'}{k \blacktriangleright s \underset{\beta,a,\rho}{\rightsquigarrow} k' \blacktriangleright s} \qquad \frac{k \underset{\beta,a,\rho}{\rightsquigarrow} k'}{k \blacktriangleleft \underset{\beta,a,\rho}{\rightsquigarrow} k' \blacktriangleleft}$$

$$\frac{k \underset{\beta,a,\rho}{\rightsquigarrow} k'}{k \triangleright i \underset{\beta,a,\rho}{\rightsquigarrow} k' \triangleright i} \qquad \frac{k \underset{\beta,a,\rho}{\rightsquigarrow} k'}{k \triangleleft v \underset{\beta,a,\rho}{\rightsquigarrow} k' \triangleleft v}$$

图 21. 动态调整线程优先级的规则。

5.2 正确性证明

本节的主要目标是证明，a λ_s^4 程序的执行所产生的图是良构的，也就是说，其中不包含优先级反转（操作语义的动态优先级天花板机制未处理的优先级反转）。我们将此证明与标准的类型保持性，以及配置和图上若干有助于证明良构性的额外不变式的保持性结合起来。首先，引理 5.1表明，通过 $K \underset{\beta,a,\rho}{\rightsquigarrow} K'$判断及其关于栈的辅助判断 $k \underset{\beta,a,\rho}{\rightsquigarrow} k'$所实现的动态优先级变更，不会破坏栈和栈状态的类型。

引理 5.1. • 若 $\vdash^R_\Sigma k \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho)$ 且 $\beta @ \rho' \in \Sigma$ 且 $k \underset{\beta,a,\rho}{\rightsquigarrow} k'$ 则 $\vdash^R_\Sigma k' \blacktriangleleft : (\Psi_1, R_1) \rightsquigarrow (\Psi_2, \rho')$

• 若 $\Psi \vdash^R_\Sigma K @ \rho \dashv \Psi'$ 且 $\beta @ \rho' \in \Sigma$ 且 $K \underset{\beta,a,\rho}{\rightsquigarrow} K'$ 则 $\Psi \vdash^R_\Sigma K' @ \rho' \dashv \Psi'$

证明。• 对 $k \underset{\beta,a,\rho}{\rightsquigarrow} k'$的推导进行归纳。– $k = k_0 :: \text{with* mutex}[\beta] -$ 和 $k' = k_0 :: \text{with*}[\beta;\ \rho]\,\text{mutex}[\beta] -$。通过逆向分析，$\vdash^R_\Sigma k_0 \blacktriangleleft : (\Psi_1, R'1) \rightsquigarrow (\Psi_2, \rho)$，其中 $R_1 = R'_1 \cup \{\rho'\}$，且 $\cdot \vdash^R_\Sigma \text{mutex}[\beta] : \text{mutex}[\rho']$。根据 KS-WithLockSP，$\vdash^R_\Sigma k' \blacktriangleleft : (R_1, \Psi_1) \rightsquigarrow (R_1, \Psi_2)\rho'$。– 由归纳假设得证。

带同步的响应式并行

- 根据第（1）部分。

□

现在我们介绍用于证明图的良构性的各项不变式。不变式 (1)就是配置具有良类型；这一不变式得到保持这一事实对应于标准的保持性结果。不变式 (2) 本身表明图是良构的。其余不变式陈述了证明该不变式得以保持所需的性质。不变式 (3)–(6) 涉及互斥锁，而不变式 (7)–(8)涉及条件变量。两个不变式直接确保了良构性的某些方面：不变式 (3) 确保从持有锁的线程到等待该锁的线程存在弱边；这是良构性定义所要求的。不变式 (8) 通过指出此类顶点没有条件变量权限（而要向它们发送信号则需要该权限），确保高优先级线程不会由于同步边而在其关键路径上接收到低优先级顶点。不变式 (3)、(5)和(6)对处理互斥锁的线程的优先级提出了要求：首先，在包含该互斥锁的每个签名中，每个互斥锁都与一个优先级相关联，即其优先级上限ρ_β；其次，每个等待互斥锁的线程的优先级必须同时低于该上限和当前持有该锁的线程的优先级。不变式 (4) 表明没有线程在等待一个未锁定的锁，这限制了我们必须在等待线程上维护这些不变式的步骤。不变式(7) 表明，任何等待条件变量的线程的优先级都低于该条件变量的句柄。最后一个不变式(9)以及(3)的最后一部分，仅为证明进展所必需；它们表明，任何以低于其上限的优先级持有锁的线程都可以提升至该优先级上限。

定义 *5 .2 (不变式).* 我们要求配置 $\mu; \sigma; W;\ L$ 满足以下不变式 $|\ g$，其中 $\mu = a_1 \underset{\rho_1\ \Sigma_1}{\hookrightarrow}$, $s^1 \otimes \ldots \otimes a_n \underset{\rho_n}{\hookrightarrow}$, $_{\Sigma_n}$ s_n。（1） $\Sigma \vdash^R \mu; \sigma; W;\ L$。下文中，我们将用 Ψ_i指代用于 $a_i \in \mu$ 的类型推导中的权限。（2） g 良构。（3）如果 $L(\beta) = (a, u_1, u_2)$ 且 $(a'u'\ \rho_{a'}, \Sigma_{a'}, K_{a'}) \in W(\beta)$，那么 $(u_1, u')^w \in g$ 且 $\rho_{a'} \preceq \mathrm{Prio}^g(a)$，并且如果 $\rho_{a'} \neq \rho_\beta$，那么 $K_{a'} \underset{\beta, a', \rho_{a'}}{\rightsquigarrow} K'$。（4）如果 $L(\beta) = \bot$，那么 $W(\beta) = []$。（5）对于所有互斥锁 β，存在一个优先级 ρ_β，使得对于所有 i，若 $\beta@\rho \in \Sigma_i$，则 $\rho = \rho_\beta$。（6）对于所有互斥锁 β，若 $(a, (u_1, u_2)\ \rho, \Sigma, K) \in W(\beta)$，则 $\rho \preceq \rho_\beta$。（7）对于所有条件变量 α 以及所有 $(a, (u_1, u_2)\ \rho_a, \Sigma_a, K'a) \in W(\alpha)$，我们有 $\alpha @\rho \in \Sigma_a$，其中 $\rho_a \preceq \rho$。（8）若 $(u'\ \Sigma') \not\sqsupseteq^s (u, \Sigma)$ 且 $\alpha@\rho \in \Sigma$，并且 a_i 中的最后一个顶点是 u' 的后代，那么

- 对于所有满足 $\rho \not\preceq \rho'$ 的 ρ'， $\Psi_i(\alpha, \rho') =$ 为 None，且
- 若 $\rho \not\preceq PRIO^g(u')$ 且 $\rho \preceq PRIO^g(a_i)$，则对于所有 ρ'， $\Psi_i(\alpha, \rho') =$ 为 None。(9) 若

$L\ \beta = a\ u_1\ u_2$ $a \underset{\rho\ \Sigma}{\hookrightarrow} K \in \mu$ $a\ u_1\ u_2\ \rho\ \Sigma\ K \in W\ \eta$ $\beta\ \rho_\beta \in \Sigma$ $\rho \neq \rho_\beta$ $K \underset{\beta, a, \rho}{\rightsquigarrow}$ ( ) (,,)，则 ,或 (,(,),,) ( ) 和 @, K 且若 ，则 ′。

定理 5.3（保持性）指出，动态语义的一步执行会保持不变式。

定理 *5.3*。假设 $\mu = a_1 \underset{\rho_1}{\hookrightarrow}$, $_{\Sigma_1}$ $s_1 \otimes \ldots \otimes a_n \underset{\rho_n}{\hookrightarrow}$, $_{\Sigma_n}$ s_n和 $\mu; \sigma; W;\ L \mid g$满足定义 *5.2* 的不变式。若 $\mu; \sigma; W;\ L \mid g \Rightarrow \mu';\ \sigma'; W';\ L' \mid g'$，则 $\mu';\ \sigma'; W';\ L' \mid g'$ 满足这些不变式。

证明。对 μ； σ； W； $L \mid g \Rightarrow \mu'$； σ'； W'； $L' \mid g'$ 的推导进行归纳。

- 生成。然后 $\mu = a \underset{\rho}{\hookrightarrow}$, $_\Sigma$ $k \triangleright$ 生成$[\rho']\{s'\} \otimes \mu_0$ 以及 $\mu' = a \underset{\rho}{\hookrightarrow}$, $_\Sigma$ $k \triangleleft$ hi $\otimes b \underset{\rho}{\hookrightarrow}$, $_\Sigma$ $\epsilon \triangleright s$ $'\mu_0$。

（1）根据对类型规则 KS-PopInstr 和 Spawn 的逆向分析，$\vdash^R_\Sigma k \lhd:\ (\text{unit}\ \Psi_2, R_1) \rightsquigarrow (\Psi'\ \rho)$ 以及 $\Psi = \Psi_1, \Psi_2$和 $\Gamma;\ \Psi_1 \vdash^R_\Sigma s'@\rho' \dashv \Psi_0$ 以及 $\forall \alpha, \rho''.\Psi(\alpha, \rho'')\ \neq \text{None}\ \rightarrow \Psi(\alpha, \rho)\ \neq \text{None}$。根据 unitI 和 KS-PushInstr，$\Psi_2 \vdash^R_\Sigma k \lhd \text{hi} @\ \rho \dashv \Psi'$。对于所有 $\alpha, \rho, \Psi(\alpha, \rho)\ \rightarrow \Psi_1(\alpha, \rho)\ \otimes \Psi_2(\alpha, \rho)$，因此权限得以保留。（2）不会创建新的同步边，因此良构性得以保留。（3）不会向 W或 L添加任何项。（4） L, W保持不变。（5） $\Sigma' b = \Sigma_a$，因此任意互斥锁的优先级得以保留。不会向 W添加任何项。（6）不会向 W添加任何项。（7）不会向 W添加任何项。（8）如果 u 满足条件（8）的要求，那么 b 中的最后一个顶点也满足要求，因为如果 $\Psi(\alpha, \rho)\ = 0$，那么 $\Psi_2(\alpha, \rho)\ = 0$。如果 ρ_Σ 是条件（8）中 CV 的优先级，并且 $\rho_\Sigma \not\preceq \rho$ 和 $\rho_\Sigma \preceq \rho'$，那么对于所有 ρ_0， $\Psi_2(\alpha, \rho_0)$ =None，因为根据假设， $\Psi(\alpha, \rho)\ = \text{None}$，所以根据类型规则前提的逆否命题， $\Psi(\alpha, \rho_0)\ = \text{None}$。（9）如果 $L(\beta)\ = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。这保持不变。• 新建引用 然后 $\mu = a \underset{\rho}{\hookrightarrow}_\Sigma\ k \rhd \text{ref}[\tau]\ y \otimes \mu_0$和 $\mu' = a \underset{\rho\ \Sigma}{\hookrightarrow}\ ,_{s\sim\tau}\ k \lhd \text{ref}[s] \otimes \mu_0$和 $\sigma' = \sigma[s \mapsto (y, u, \Sigma)]$。（1）根据逆向分析， $\Gamma;\ \Psi \vdash^R_\Sigma \text{ref}[\tau]\ \sim:\ \tau\text{ref}@\rho \dashv \Psi$ 以及 $\vdash^R_\Sigma k \lhd:\ (\tau\text{ref}\ \Psi, R_1) \rightsquigarrow (\Psi',)$ 和 $\Gamma \vdash^R_\Sigma y:\ \tau$。根据 RevVal， $\Gamma \vdash^R_{\Sigma, s\sim\tau} \text{ref}[s]:\ \tau\text{ref}$。我们有 $\Sigma, s\sim\tau \vdash \sigma'$。根据 KS-PushInstr， $\Psi \vdash^R_\Sigma k \lhd \text{ref}[s] @\ \rho \dashv \Psi'$。（2）不会创建新的同步边，因此良构性得以保留。（3）不会向 W或 L添加任何项。（4） L, W 保持不变。（5） s 不是互斥锁，因此该条件得以保留。（6）不会向 W添加任何项。（7）不会向 W添加任何项。（8）此步骤不会拆分或重新分配权限。（9）如果 $L(\beta)\ = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。这保持不变。• 解引用 然后 $\mu = a \underset{\rho}{\hookrightarrow}_\Sigma\ k \rhd !\text{ref}[s] \otimes \mu_0$ 和 $\mu' = a \underset{\rho\ \Sigma}{\hookrightarrow}\ ,_{\Sigma'}\ k \lhd v \otimes \mu_0$其中 $\sigma(s)\ = (v, u'\ \Sigma')$。（1）根据逆向分析， $\Gamma;\ \Psi \vdash^R_\Sigma !\text{ref}[s] \sim:\ \tau@\rho \dashv \Psi$ 以及 $s\sim\tau \in \Sigma$ 和 $\vdash^R_\Sigma k \lhd:\ (\tau, \Psi, R_1) \rightsquigarrow (\Psi'\ \rho)$。由于 $\Sigma \vdash \sigma$，我们有 $\Gamma \vdash^R_\Sigma v:\ \tau$。根据 KS-PushInstr， $\Psi \vdash^R_\Sigma k \lhd v @\ \rho \dashv \Psi'$。（2）不会创建新的同步边，因此良构性得以保留。（3）不会向 W或 L添加任何项。（4） L, W 保持不变。（5） Σ 和 Σ'满足该条件，因此 Σ, Σ'也满足。（6）不会向 W添加任何项。（7）不会向 W添加任何项。（8）此步骤不会拆分或重新分配权限。（9）如果 $L(\beta)\ = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。这保持不变。• 更新 然后 $\mu = a \underset{\rho}{\hookrightarrow}_\Sigma\ k \rhd \text{ref}[s] := y \otimes \mu_0$ 以及 $\mu' = a \underset{\rho}{\hookrightarrow}_\Sigma\ k \lhd \text{hi}\ \otimes \mu_0$和 $\sigma' = \sigma[s \mapsto (y, u, \Sigma)]$。

带同步的响应式并行

(1) 通过逆向分析， $\Gamma; \Psi \vdash_{\Sigma}^{R} \mathrm{ref}[s] := y \sim: \mathrm{unit}@\rho \dashv \Psi$ and $\Gamma \vdash_{\Sigma}^{R} \mathrm{ref}[s]: \tau\ \mathrm{ref}$ and $\Gamma \vdash_{\Sigma}^{R} y: \tau$ and $\vdash_{\Sigma}^{R} k \lhd: (\mathrm{unit}\ \Psi, R_1) \rightsquigarrow (\Psi'\ \rho)$。根据 KS-PushInstr， $\Psi \vdash_{\Sigma}^{R} k \lhd \mathrm{hi}\ @\ \rho \dashv \Psi'$。我们仍有 $\Sigma \vdash \sigma'$，因为 y 的类型为 τ。(2) 没有创建新的同步边，因此良构性得以保持。(3) 没有向 W 或 L 添加任何项。(4) L, W 保持不变。(5) 签名未改变。(6) 没有向 W 添加任何项。(7) 没有向 W 添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^p，有 $k \underset{\beta, a, \rho}{\rightsquigarrow} k^p$。这一点保持不变。• NewCV 然后 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \rhd \mathrm{newcv}[\rho'] \otimes \mu_0$ and $\mu' = a \underset{\rho\ \Sigma\ \alpha}{\hookrightarrow}, , @\rho' k \lhd \mathrm{cv}[\alpha] \otimes \mu_0$。(1) 通过逆向分析， $\Gamma; \Psi \vdash_{\Sigma}^{R} \mathrm{newcv}[\rho'] \sim: \mathrm{cv}[\alpha;\ \rho']@\rho \dashv \Psi'$ and $\vdash_{\Sigma}^{R} k \lhd: (\mathrm{cv}[\alpha;\ \rho'], \Psi' R_1) \rightsquigarrow (\Psi''\ \rho)$。根据 CV， $\Gamma \vdash_{\Sigma,\alpha}^{R} @\rho'\ \mathrm{cv}[\alpha]: \mathrm{cv}[\alpha; \rho']$。根据 KS-PushInstr， $\Psi' \vdash_{\Sigma}^{R} k \lhd \mathrm{cv}[\alpha]\ @\ \rho \dashv \Psi''$。(2) 没有创建新的同步边，因此良构性得以保持。(3) 没有向 W 或 L 添加任何项。(4) L, W 保持不变。(5) s 不是互斥锁，因此此性质得以保持。(6) 没有向 W 添加任何项。(7) 没有向 W 添加任何项。(8) 对所有满足 $\rho' \not\leq \rho_0$ 的 ρ_0，根据类型规则 NewCV，我们有 $\Psi'(\alpha, \rho_0) = \mathrm{None}$。顶点 u 没有真后代，因此，如果 a_i 的最后一个顶点是 u 的后代，那么 $a_i = a$ 且 $\mathrm{Prio}_{\mathcal{G}'}(u) = \mathrm{Prio}_{\mathcal{G}'}(a_i)$，这与不变式 8 第二部分的前提 $\rho \not\leq \mathrm{Prio}_{\mathcal{G}'}(u)$ and $\rho \leq \mathrm{Prio}_{\mathcal{G}'}(a_i)$ 矛盾。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。这一点保持不变。• Wait 然后 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \rhd \mathrm{wait}\ \mathrm{cv}[\alpha] \otimes \mu'$ and $W' = W[\alpha \mapsto (a, (u_1, u_2)\ \rho, \Sigma, [\mathrm{hi}/x]s) :: W(\alpha)]$。(1) 通过逆向分析， $\Gamma; \Psi \vdash_{\Sigma}^{R} \mathrm{wait}\ \mathrm{cv}[\alpha] \sim: \mathrm{unit}@\rho \dashv \Psi$ and $\vdash_{\Sigma}^{R} k \lhd: (\mathrm{unit}\ \Psi, R_1) \rightsquigarrow (\Psi'\ \rho)$。根据 KS-PushInstr， $\Psi \vdash_{\Sigma}^{R} k \lhd \mathrm{hi}\ @\ \rho \dashv \Psi'$。(2) 没有创建新的弱边或同步边，因此良构性得以保持。(3) α 不是互斥锁，因此 $\alpha \notin \mathrm{Dom}(L)$。(4) α 不是互斥锁。(5) α 不是互斥锁，因此此条件不适用。(6) α 不是互斥锁，因此此条件不适用。(7) 根据对 Wait 和 CV 的逆向分析， $\alpha@\rho' \in \Sigma$，其中 $\Gamma \vdash^{R} \rho \leq \rho'$。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。这一点保持不变，不过 a 从 μ 移至 $W'(\beta)$。

• Signal 1 $\mu = a \underset{\rho\ \Sigma}{\hookrightarrow} k \rhd [\alpha] \otimes \mu_0$ $\mu = a \underset{\rho\ \Sigma}{\hookrightarrow} k \lhd \otimes b \underset{\rho_b\ \Sigma_b}{\hookrightarrow} K_b \mu_0$ b _ u ρ_b Σ_b K_b $\in$ 然后，信号 cv 和 $\prime$，$\overset{W}{\mathrm{hi}}$，$\overset{\alpha}{\mathrm{hi}}$ 以及 (, (, $\prime$),,) ( )，$\overset{\Psi}{\mathrm{hi}}$ $\overset{\alpha}{\mathrm{hi}}$ $\overset{\neq}{\mathrm{hi}}$，并通过逆向分析，(, ρ) None。

(1) 通过逆向分析，$\Psi_b \vdash^R_{\Sigma_b} K_b @\ \rho_b \dashv \Psi'_b$和 Γ；$\Psi \vdash^R_\Sigma$信号 $\mathrm{cv}[\alpha] \rightsquigarrow: \mathrm{unit}@\rho \dashv \Psi$和 $\vdash^R_\Sigma k \lhd: (\mathrm{unit}\ \Psi, R_1) \rightsquigarrow (\Psi'\ \rho)$。根据 KS-PushInstr，$\Psi \vdash^R_\Sigma k \lhd \mathrm{hi} @\ \rho \dashv \Psi'$。(2) 我们需要证明边 (u, u') 不违反良构性。只需证明，对于 u 的所有强祖先 u_0，$\rho_b \preceq \mathrm{Prio}_{g'}(u_0)$。根据假设，$\alpha@\rho_\alpha \in \Sigma_b$和 $\rho_b \preceq \rho_\alpha$。如果是 $u_0 \sqsupseteq^s u'$，那么根据良构性，这一点必定已经成立。如果是 $u_0 \not\sqsupseteq^s u'$，那么 $\rho_\alpha \preceq \mathrm{Prio}_{g'}(u_0)$，否则 $\Psi(\alpha, \rho) = \mathrm{None}$。(3) 没有项目添加到 W 或 L。(4) α 不是互斥锁，因此条件不变。(5) α 不是互斥锁，因此此条件不适用。(6) α 不是互斥锁，因此此条件不适用。(7) 没有项目添加到 W。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。这一点保持不变。• Signal2。与 Signal1 类似。

• 提升 Then $\mu = a \underset{\rho}{\hookrightarrow}_\Sigma k \rhd$ 将 $\mathrm{cv}[\alpha]$ 提升为 $\rho' \otimes \mu_0$ 和 $\mu' = a \underset{\rho\ \Sigma\ \alpha\ \rho}{\hookrightarrow}, , @'\ k \lhd \mathrm{cv}[\alpha] \otimes \mu_0$. (1) 通过逆向分析，$\Gamma$; $\Psi \vdash^R_\Sigma$将 $\mathrm{cv}[\alpha]$ 提升为 $\rho' \rightsquigarrow: \mathrm{cv}[\alpha;\ \rho']@\rho \dashv \Psi'$ and $\vdash^R_\Sigma k \lhd: (\mathrm{cv}[\alpha;\ \rho'], \Psi' R_1) \rightsquigarrow (\Psi''\ \rho)$ and $\forall \rho_0 . \rho_2 \not\preceq \rho_0 \to \Psi(\alpha, \rho_0) =$ 独占 and $\forall \rho_0 . \rho_2 \not\preceq \rho_0 \to \Psi'(\alpha, \rho_0) = \mathrm{None}$ and $\forall \rho_0 . \rho_2 \preceq \rho_0 \to \Psi'(\alpha, \rho_0) = \Psi(\alpha, \rho_0)$. 根据条件变量和 KS-PushInstr，$\Psi' \vdash^R_{\Sigma, \alpha@\rho'} k \lhd \mathrm{cv}[\alpha] @\ \rho \dashv \Psi''$. (2) 不会创建新的弱边或同步边，因此良构性得以保留。(3) 没有向 W或 L中添加任何项。(4) α不是互斥锁，因此这一条件不变。(5) α不是互斥锁，因此此条件不适用。(6) α不是互斥锁，因此此条件不适用。(7) 没有向 W中添加任何项。(8) 现在，$\alpha@\rho'$位于 Σ中，因此我们必须检查所有满足 $u' \not\sqsupseteq^s u$ 的 u'的条件。设 u_e是 u'的后代，同时也是 g 中某个线程的最后一个顶点。如果 $u_e = u$，则其优先级将为 ρ，且类型为Ψ''。对于所有满足 $\rho' \not\preceq \rho''$ 的 ρ''，都有 $\Psi''(\alpha, \rho'') = \mathrm{None}$，符合要求。如果 u_e不是 u的后代，那么对于任意满足 $\rho' \not\preceq \rho''$ 的优先级 ρ''，它在 α 上必须具有 None 权限，因为 $\Psi(\alpha, \rho'') =$ 独占。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta\ a\ \rho}{\rightsquigarrow}, k^p$。这一点保持不变。• NewMutex Then $\mu = a \underset{\rho}{\hookrightarrow}_\Sigma k \rhd$ 新建互斥锁$[\rho'] \otimes \mu_0$ and $\mu' = a \underset{\rho\ \Sigma\ \beta\ \rho}{\hookrightarrow}, , @'\ k \lhd$ 互斥锁$[\beta] \otimes \mu_0$. (1) 通过逆向分析，Γ; $\Psi \vdash^R_\Sigma$新建互斥锁$[\rho'] \rightsquigarrow:$ 互斥锁$[\rho']@\rho \dashv \Psi$ and $\vdash^R_\Sigma k \lhd:$互斥锁$[\rho'], \Psi, R_1) \rightsquigarrow (\Psi'\ \rho)$。根据 Mutex 和 KS-PushInstr，$\Psi \vdash^R_{\Sigma, \beta@\rho'} k \lhd$互斥锁$[\beta] @\ \rho \dashv \Psi'$。(2) 不会创建新的同步边，因此良构性得以保留。(3) 没有向 W或 L中添加任何项。(4) L, W保持不变。(5) β是新鲜的，因此 $\beta@\rho'$是 β唯一的条目。(6) 没有向 W中添加任何项。(7) 没有向 W中添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta\ a}{\rightsquigarrow}, k^p$。这一点保持不变。

带同步的响应式并行

• Let1 Then $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright$ 令 $x = i$ 于 $sv \otimes \mu_0$ 中，并且 $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k ::$ let $x = -$ in $s \triangleright i \otimes \mu_0$. (1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi_3, R_1) \rightsquigarrow (\Psi_4, \rho)$，且对于所有 $\rho' \in R_1$，我们有 $\cdot ; \Psi \vdash^R_\Sigma i \sim : \tau @ \rho' \dashv \Psi_2$ and $x : \tau ; \Psi_2 \vdash^R_\Sigma s @ \rho' \dashv \Psi_3$。根据 KS-Let，$\vdash^R_\Sigma k :: \text{let } x = - \text{ in } s \triangleleft : (\tau, \Psi_2, R_1) \rightsquigarrow (\Psi_4, \rho)$。根据 KS-PopInstr，$\Psi \vdash^R_\Sigma k :: \text{let } x = - \text{ in } s \triangleright i @ \rho \dashv \Psi_4$。(2) 不会创建新的同步边，因此良构性得以保留。(3) 没有向 W 或 L中添加任何项。(4) L 保持不变。(5) 签名没有改变。(6) 没有向 W中添加任何项。(7) 没有向 W中添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^ρ，我们有 $k \underset{\beta, a, \rho}{\rightsquigarrow} k^\rho$。这一点保持不变。.

• Let2 Then $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k :: \text{let } x = - \text{ in } s \triangleleft v \otimes \mu_0$ and $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright [v/x]s \otimes \mu_0$. (1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi' \rho)$ and $\cdot \vdash^R_\Sigma v : \tau$ and 对所有 $\rho' \in R_1$，我们有 $x : \tau ; \Psi \vdash^R_\Sigma s @ \rho' \dashv \Psi''$。通过替换，$\cdot ; \Psi \vdash^R_\Sigma [v/x]s @ \rho' \dashv \Psi''$。根据 KS-PopStmt，$\Psi \vdash^R_\Sigma k \blacktriangleright [v/x]s @ \rho \dashv \Psi'$。(2) 不会创建新的同步边，因此良构性得以保持。(3) 不会向 W或 L添加任何项。(4) L, W保持不变。(5) 签名不会改变。(6) 不会向 W添加任何项。(7) 不会向 W添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^ρ，我们有 $k \underset{\beta, a, \rho}{\rightsquigarrow} k^\rho$。这一点保持不变。.

• WithLockS1 接着 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright$ 带互斥锁$[\beta]$ $s \otimes \mu_0$ 和 $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k ::$ 带*互斥锁$[\beta] - \blacktriangleright s \otimes \mu_0$。以及 $L' = L[\beta \mapsto (a, u_1, u_2)]$。(1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi' \rho)$，且对于所有 $\rho' \in R_1$，我们有 $\cdot ; \Psi \vdash^R_\Sigma$带互斥锁$[\beta]$ $s @ \rho' \dashv \Psi''$。通过对 WithLock 进行逆向分析，我们有 $\beta @ \rho_\beta \in \Sigma$和 $\cdot ; \Psi \vdash^R_\Sigma s @ \rho' \dashv \Psi''$，以及 $\cdot ; \Psi \vdash^R_\Sigma s @ \rho_\beta \dashv \Psi''$。根据 KS- WithLockS，$\vdash^R_\Sigma k ::$ 带*互斥锁$[\beta] - \blacktriangleleft : (\Psi'' R_1 \cup \{\rho\beta\}) \rightsquigarrow (\Psi' \rho)$。根据 Seq 和 KS-PopStmt，$\Psi \vdash^R_\Sigma k ::$ 带*互斥锁$[\beta] - \blacktriangleright s @ \rho \dashv \Psi'$。(2) 没有添加同步边，因此良构性得以保持。(3) 根据假设，$W(\beta) = []$。(4) $L'(\beta) \neq \bot$。(5) 签名未发生变化。(6) 未向 W添加任何项。(7) β不是条件变量，因此不变式得以保持。(8) 此步骤不会拆分或重新分配权限。

(9) 假设 $L(\beta\prime) = (a, \_, \_)$。如果 $\beta = \beta\prime$，那么我们有

$$k :: \texttt{with*}\ \texttt{mutex}[\beta] - \underset{\beta,a,\rho}{\rightsquigarrow} \texttt{with*}[a;\rho]\ \texttt{mutex}[\beta] -$$

否则，不变式未发生变化。

- WithLockS2 Then $\mu = a \overset{\rho}{\hookrightarrow}, \Sigma$ $k \blacktriangleright$ 带互斥锁$[\beta]$ $s \otimes \mu'$ 和 $W' = W[\beta \mapsto (a, (u_1, u_2)\ \rho, \Sigma, K') :: W(\beta)]$其中 $K' = k ::$ 带*互斥锁$[\beta] - \blacktriangleright s$和 $L(\beta) = (a'\, u'1, u'2)$ 以及 $\rho \preceq \text{Prio}_g(a')$。

（1）通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi'\, \rho)$，并且对于所有 $\rho' \in R_1$，我们有 $\cdot; \Psi \vdash^R_\Sigma$with 互斥锁$[\beta]$ $s @ \rho' \dashv \Psi''$。根据对 WithLock 的逆向分析，我们有 $\beta @ \rho_\beta \in \Sigma$ 和 $\cdot; \Psi \vdash^R_\Sigma s @ \rho' \dashv \Psi''$，以及 $\cdot; \Psi \vdash^R_\Sigma s @ \rho_\beta \dashv \Psi''$。根据 KS- WithLockS，$\vdash^R_\Sigma k ::$ with* 互斥锁$[\beta] - \blacktriangleleft : (\Psi'' R_1 \cup \{\rho\beta\}) \rightsquigarrow (\Psi'\, \rho)$。根据 Seq 和 KS-PopStmt，$\Psi \vdash^R_\Sigma k ::$ with* 互斥锁$[\beta] - \blacktriangleright s @ \rho \dashv \Psi'$。

(2) 未添加同步边，因此良构性得以保持。

(3) 根据假设，对于所有 $(c, (u\prime 1, u\prime 2)\ \rho_c, \Sigma_c, s_c) \in W(\beta)$，都有一条弱边 $(u\prime 1, u\prime 2)$，且 $\rho_c \preceq Prio_g(a\prime)$。我们向 $g\prime$ 添加一条弱边 $(u\prime 1, u_2)$，因此弱边上的不变式得以保持。由于 $\rho \preceq Prio_g(a\prime)$，优先级上的不变式也得以保持。我们有

$$k :: \texttt{with*}\ \texttt{mutex}[\beta] - \underset{\beta,a,\rho}{\rightsquigarrow} \texttt{with*}[a;\rho]\ \texttt{mutex}[\beta] -$$

因此，这项要求也得以保持。

(4) $L\prime(\beta) \neq \bot$.

(5) 签名没有改变。

(6) 对 WithLock 和 Mutex 进行逆向分析，$\beta @ \rho_\beta \in \Sigma$ and $\rho \preceq \rho_\beta$。

(7) β 不是条件变量，因此不变式得以保持。

(8) 此步骤不会拆分或重新分配权限。

(9) 如果 $L(\beta\prime) = (a, \_, \_)$，那么我们有 $k \underset{\beta', a,}{\rightsquigarrow} \quad k^p$ 对于某个 k^p。这保持不变。ρ

- WithLockS3 然后 $\mu = a \overset{\rho}{\hookrightarrow}, \Sigma$ $k \blacktriangleright$使用互斥锁$[\beta]$ $s \otimes b \overset{\rho\ \Sigma}{\hookrightarrow}, '$ $K_b \otimes \mu \otimes \mu_0$和 $\mu' = b' \overset{\rho_\beta\ \Sigma}{\hookrightarrow}, '$ $K'' \otimes \mu_0$和 $W' = W[\beta \mapsto (a, (u_1, u_2)\ \rho, \Sigma, K') :: W(\beta)]$其中 $K' = k ::$ with* 互斥锁$[\beta] - \blacktriangleright s$和 $L(\beta) = (a'\, u'1, u'2)$。

（1）通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi'\, \rho)$，并且对于所有 $\rho' \in R_1$，都有 $\cdot;\ \Psi \vdash^R_\Sigma$使用互斥锁$[\beta]$ $s @ \rho' \dashv \Psi''$。通过对 WithLock 进行逆向分析，我们有 $\beta @ \rho_\beta \in \Sigma$，以及 $\cdot;\ \Psi \vdash^R_\Sigma s @ \rho' \dashv \Psi''$ 和 $\cdot;\ \Psi \vdash^R_\Sigma s @ \rho_\beta \dashv \Psi''$。根据 KS-WithLockS，$\vdash^R_\Sigma k ::$ with* 互斥锁$[\beta] - \blacktriangleleft : (\Psi'' R_1 \cup \{\rho\beta\}) \rightsquigarrow (\Psi'\, \rho)$。根据 Seq 和 KS-PopStmt，$\Psi \vdash^R_\Sigma k ::$ with* 互斥锁$[\beta] - \blacktriangleright s @ \rho \dashv \Psi'$。同样通过逆向分析，$\Psi_b \vdash^R_{\Sigma'} K_b @ \rho' \dashv \Psi' b$。根据引理 5.1，$\Psi_b \vdash^R_{\Sigma'} K'' @ \rho_\beta \dashv \Psi' b$。

（2）没有添加同步边，因此良构性得以保持。

（3）根据假设，对于所有 $(c,\ (u_{1c},\ u_{2c})\ \rho_c,\ \Sigma_c,\ s_c) \in W(\beta)$，$\rho_c \preceq PRIO_g(a\prime) \overset{\preceq}{\rightsquigarrow} \rho_\beta$。通过逆向分析，$\rho \preceq \rho_\beta$。我们添加来自 $u\prime 1$ 的所有相关弱边。我们有 $k ::$ with* 互斥锁$[\beta] - \beta,a,\rho$ with*$[a;\rho]$ 互斥锁$[\beta] -$，因此这一要求也得以保持。

（4） $L'(\beta) \neq \bot$.

（5）签名不会改变。

（6）对 WithLock 和 Mutex 进行逆向分析，$\beta @ \rho_\beta \in \Sigma$ 和 $\rho \preceq \rho_\beta$。

（7） β 不是条件变量，因此不变式得以保持。

(8) 此步骤不会拆分或重新分配权限。

带同步的响应式并行

(9) 如果 $L(\beta_\prime) = (a, \_, \_)$，那么对于某个 k^ρ，我们有 $k \underset{\beta',a,\rho}{\rightsquigarrow} k^\rho$。这一点保持不变。$b_\prime$ 的优先级为 ρ_β，因此此属性不再适用于 $b_\prime$。

• WithLockS4 然后 $\overset{\mu = a}{\underset{\rho}{\hookrightarrow}}_{,\Sigma}\ k\blacktriangleright$ 使用互斥锁 $[\beta]\ s \otimes \mu \otimes \mu'$ 和 $W' = W[\eta' \mapsto \ell_1 :: (b' \_, \rho_\beta, \Sigma' K'') :: \ell_2][\beta \mapsto (a, (u_1, u_2)\ \rho, \Sigma, K') :: W(\beta)]$ 其中 $W(\eta') = \ell_1 :: (b, \_, \rho' \Sigma' K_b) :: \ell_2$ 且 $K' = k ::$ with* 互斥锁 $[\beta] - \blacktriangleright s$ 且 $L(\beta) = (a' u' 1, u' 2)$. (1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi' \rho)$，并且对于所有 $\rho' \in R_1$，我们有 $\cdot; \Psi \vdash^R_\Sigma$ 使用互斥锁 $[\beta]\ s@\rho' \dashv \Psi''$ 通过对 WithLock 进行逆向分析，我们有 $\beta@\rho_\beta \in \Sigma$ 且 $\cdot; \Psi \vdash^R_\Sigma s@\rho' \dashv \Psi''$，以及 $\cdot; \Psi \vdash^R_\Sigma s@\rho_\beta \dashv \Psi''$。根据 KS- WithLockS，$\vdash^R_\Sigma k ::$ with* 互斥锁 $[\beta] - \blacktriangleleft : (\Psi'' R_1 \cup \{\rho_\beta\}) \rightsquigarrow (\Psi' \rho)$。根据 Seq 和 KS-PopStmt，$\Psi \vdash^R_\Sigma k ::$ with* 互斥锁 $[\beta] - \blacktriangleright s@\ \rho \dashv \Psi'$。此外，通过逆向分析，$\Psi_b \vdash^R_\Sigma K_b@\ \rho' \dashv \Psi' b$。根据引理 5.1，$\Psi_b \vdash^R_\Sigma K''\ @\ \rho_\beta \dashv \Psi' b$。(2) 没有添加同步边，因此良构性得以保持。(3) 根据假设，对于所有 $(c, (u_{1c}, u_{2c})\ \rho_c, \Sigma_c, s_c) \in W(\beta)$，$\rho_c \preceq \text{Prio}_g(\underset{\rightsquigarrow}{a'}) \preceq \rho_\beta$。通过逆向分析，$\rho \preceq \rho_\beta$。我们添加来自 u'' 1 的所有相关弱边。我们有 $k ::$ with* 互斥锁 $[\beta] - \beta,a,\rho$ with*$[a; \rho]$ 互斥锁 $[\beta] -$，因此这一要求也得以保持。(4) $L'(\beta) \neq \bot$。(5) 签名未发生变化。(6) 通过对 WithLock 和 Mutex 进行逆向分析，$\beta@\rho_\beta \in \Sigma$ 且 $\rho \preceq \rho_\beta$。

（7）。β 不是条件变量，因此不变式得以保持。（8）此步骤不会拆分或重新分配权限。（9）如果 $L(\beta_\prime) = (a, \_, \_)$，那么对于某个 k^ρ，我们有 $k \underset{\beta',a,\rho}{\rightsquigarrow} k^\rho$。这一点保持不变。$b_\prime$ 的优先级为 ρ_β，因此该属性不再适用于 $b_\prime$。

• WithLockE1 然后 $\overset{\mu = a}{\underset{\rho}{\hookrightarrow}}_{,\Sigma}\ k ::$ with* 互斥锁 $[\beta] - \blacktriangleleft \otimes \mu_0$ 和 $\overset{\mu' = a}{\underset{\rho}{\hookrightarrow}}_{,\Sigma}\ k \blacktriangleleft \otimes \mu_0$。(1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi, R_1) \rightsquigarrow (\Psi' \rho)$。根据 KS-PushStmt，$\Psi \vdash^R_\Sigma k \blacktriangleleft @ \rho \dashv \Psi'$。(2) 没有添加同步边，因此良构性得以保持。(3) $L'(\beta) = \bot$。(4) $W' = W$ 并且根据假设，$W(\beta) = []$。(5) 签名未发生变化。(6) 没有向 W 添加任何项。(7) β 不是条件变量，因此不变式得以保持。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^ρ，有 $k \underset{\beta\ a}{\rightsquigarrow}_{,\rho} k^\rho$。此情况保持不变。• WithLockE2 然后 $\overset{\mu = a}{\underset{\rho}{\hookrightarrow}}_{,\Sigma}\ k ::$ with* 互斥锁 $[\beta] - \blacktriangleleft \otimes \mu_0$ 和 $\overset{\mu' = a}{\underset{\rho}{\hookrightarrow}}_{,\Sigma}\ k \blacktriangleleft \otimes b \underset{\rho_b}{\longhookrightarrow}_{,\Sigma_b} K_b \otimes \mu_0$。(1) 通过逆向分析，

$\vdash^R_\Sigma k \blacktriangleleft\ \Psi\ R_1 \rightsquigarrow \Psi\ \rho$　　$\Psi \vdash^R_\Sigma k \blacktriangleleft\ \ \rho \dashv \Psi$　　$\Psi_b \vdash^R_{\Sigma_b} K_b\ \ \rho_b \dashv : (\ ,\ )\ \Psi\ (\ ')$。根据 KS-PushStmt，$@\ '$。我们还有 $@\ 'b$。(2) 添加同步边后，我们现在有 $u \sqsupseteq^s u_2$。通过对 Unlock 和 Mutex 进行逆向分析，$\beta@\rho \in \Sigma$。根据不变式 6，$\rho_b \preceq \rho$。如果 $u_0 \sqsupseteq^s u$ 并且 u_0 不是 u 的线程起点的祖先，那么根据良构性，$\text{Prio}_{g'}(u) \preceq \text{Prio}_{g'}(u_0)$，因此 $\rho_b \preceq \text{Prio}_{g'}(u_0)$。这满足了良构性的第一个组成部分。我们还有强边 (u_a, u_b)，其中 $L(\beta) = (a, u_a, u_b)$，且根据假设，$u_a \sqsupseteq^w u_2$。同样根据假设，$(u_a, u_2)^w \in g$，因此这满足良构性的第二个组成部分。

(3) $L'(\beta) = (b, u_1, u_2)$，并且我们将所有必要的边添加到 g'。根据假设，$\forall(\_, \_, \rho_c, \_, \_) \in \ell_1\ell_2 = W'(\beta)\ \rho_c \preceq \rho_b$，因此优先级不变式得以保持。(4) $L'(\beta) \neq \bot$。(5) 签名未发生变化。(6) 没有向 W 添加任何项。(7) β 不是条件变量，因此不变式得以保持。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L'(\beta') = (a, \_, \_)$，那么 $\beta' \neq \beta$。对于某个 k^p，有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。此情况保持不变。对于 $L'(\beta) = (b, u_1, u_2)$ 的情况，根据不变式 3，有 $K_b \underset{\beta', a, \rho}{\rightsquigarrow} K^p$。• WithLockE3 然后 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k$ :: with*⟦a'; ρ'⟧ 互斥锁⟦β⟧ − ◀ $\otimes\mu_0$ 和 $\mu' = a' \underset{\rho'}{\hookrightarrow}_{,\Sigma} k \blacktriangleleft \otimes\mu_0$。(1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi, R_1) \rightsquigarrow (\Psi'\ \rho')$。根据 KS-PushStmt，$\Psi \vdash^R_\Sigma k \blacktriangleleft @\rho' \dashv \Psi'$。(2) 边 (u, u') 保持良构性，因为根据假设，$\text{Prio}_{g'}(u') = \rho' \preceq \rho = \text{Prio}_{g'}(u)$。(3) $L'(\beta) = \bot$。(4) $W' = W$ 并且根据假设，$W(\beta) = []$。(5) 签名未发生变化。(6) 没有向 W 添加任何项。(7) β 不是条件变量，因此不变式得以保持。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta') = (a, \_, \_)$，那么对于某个 k^p，有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。此情况保持不变。• WithLockE4 然后 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k$ :: with*⟦a'; ρ'⟧ 互斥锁⟦β⟧ − ◀ $\otimes\mu_0$ 和 $\mu' = a' \underset{\rho'}{\overset{a,\rho}{\hookrightarrow}}_{,\Sigma} k \blacktriangleleft \otimes b \underset{\rho_b}{\hookrightarrow}_{,\Sigma_b} K_b \otimes \mu_0$。(1) 通过逆向分析，

$$\vdash^R_\Sigma k \blacktriangleleft \ \Psi\ R_1 \ \rightsquigarrow \ \Psi\ \ \rho \qquad\qquad \Psi \vdash^R_\Sigma k \blacktriangleleft \ \ \rho \ \dashv \Psi \qquad\qquad \Psi_b \vdash^R_{\Sigma_b} : (\ ,\ )$$

$K_b \quad \rho_b \dashv \Psi$

(′ ′)。根据 KS-PushStmt，@ ′ ′。我们还有 @ $'_b$。(2) 添加同步边 (u, u_2) 后，我们现在有 $u \sqsupseteq^s u_2$。通过对 Unlock 和 Mutex 进行逆向分析，$\beta@\rho \in \Sigma$。根据假设，$\rho_b \preceq \rho$。如果 $u_0 \sqsupseteq^s u$ 并且 u_0 不是 u 的线程起点的祖先，那么根据良构性，$\text{Prio}_{g'}(u) \preceq \text{Prio}_{g'}(u_0)$，因此 $\rho_b \preceq \text{Prio}_{g'}(u_0)$。这满足了良构性的第一个组成部分。边 (u, u') 也保持不变式，因为根据假设，$\text{Prio}_{g'}(u') = \rho' \preceq \rho = \text{Prio}_{g'}(u)$，而且对 u 的强祖先也适用同样的论证。我们还有强边 (u_a, u_b)，其中 $L(\beta) = (a, u_a, u_b)$，且根据假设，$u_a \sqsupseteq^w u_2$。同样根据假设，$(u_a, u_2)^w \in g$，因此这满足良构性的第二个组成部分。(3) $L'(\beta) = (b, u_1, u_2)$，并且我们将所有必要的边添加到 g'。根据假设，$\forall(\_, \_, \rho_c, \_, \_) \in \ell_1\ell_2 = W'(\beta)\ \rho_c \preceq \rho_b$，因此优先级不变式得以保持。(4) $L'(\beta) \neq \bot$。(5) 签名未发生变化。(6) 没有向 W 添加任何项。(7) β 不是条件变量，因此不变式得以保持。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L'(\beta') = (a, \_, \_)$，那么 $\beta' \neq \beta$。对于某个 k^p，有 $k \underset{\beta', a, \rho}{\rightsquigarrow} k^p$。此情况保持不变。对于 $L'(\beta) = (b, u_1, u_2)$ 的情况，根据假设，有 $K_b \underset{\beta', a, \rho}{\rightsquigarrow} K^p$。

带同步的响应式并行

• If1. 那么 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright$ 若n则 s_1 否则 $s_2 \otimes \mu_0$ 且 $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright s_1 \otimes \mu_0$。(1) 通过逆向分析，$\vdash^R_\Sigma k \blacktriangleleft: (\Psi'' R_1) \rightsquigarrow (\Psi' \rho)$，并且对于所有 $\rho' \in R_1$，我们有 $\cdot; \Psi \vdash^R_\Sigma$ 若 n 则 s_1 否则 $s_2 @ \rho' \perp \Psi''$。对 If 进行逆向分析，$\cdot; \Psi \vdash^R_\Sigma s_1 @ \rho' \dashv \Psi''$。根据 KS-PopStmt，$\Psi \vdash^R_\Sigma k \blacktriangleright s_1 @ \ \rho \dashv \Psi'$。(2) 没有添加同步边，因此良构性得以保持。(3) 没有向 W或 L添加任何项。(4) L, W未发生变化。(5) 没有更改任何签名。(6) 没有向 W添加任何项。(7) 没有向 W添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。这一点保持不变。• If2. 与 If1 类似。

• While。于是 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright$ while $v\ s \otimes \mu_0$ 和 $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright$ if v then (s; while $v\ s$) else skip $\otimes \mu_0$。(1) 根据逆向分析，$\vdash^R_\Sigma k \blacktriangleleft: (\Psi, R_1) \rightsquigarrow (\Psi' \rho)$，且对所有 $\rho' \in R_1$，我们有 $\cdot; \Psi \vdash^R_\Sigma$while $v\ s @ \rho' \dashv \Psi$。根据 While 的逆向分析，$\cdot \vdash^R_\Sigma v : \mathrm{nat}$ 且 $\cdot; \Psi \vdash^R_\Sigma s @ \rho \dashv \Psi$。根据 Seq，$\cdot; \Psi \vdash^R_\Sigma s$; while $v\ s @ \rho \dashv \Psi$。根据 Skip 和 If，$\cdot; \Psi \vdash^R_\Sigma$if $v$then (s; while $v\ s$) else skip$@\rho \dashv \Psi$。根据 KS-PopStmt，$\Psi \vdash^R_\Sigma k \blacktriangleright$if $v$then (s; while $v\ s$) else skip$@\rho \dashv \Psi'$。(2) 未添加同步边，因此良构性得以保留。(3) 未向 W或 L添加任何项。(4) L, W未改变。(5) 未更改任何签名。(6) 未向 W添加任何项。(7) 未向 W添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。此项保持不变。• Skip。(1) 根据逆向分析，$\vdash^R_\Sigma k \blacktriangleleft: (\Psi, R_1) \rightsquigarrow (\Psi' \rho)$。根据 KS-PushStmt，$\Psi \vdash^R_\Sigma k \blacktriangleleft @ \rho \dashv \Psi'$。(2) 未添加同步边，因此良构性得以保留。(3) 未向 W或 L添加任何项。(4) L, W未改变。(5) 未更改任何签名。(6) 未向 W添加任何项。(7) 未向 W添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 如果 $L(\beta) = (a, \_, \_)$，那么对于某个 k^p，我们有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。此项保持不变。• Seq1。于是 $\mu = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright s_1$; $s_2 \otimes \mu_0$ 和 $\mu' = a \underset{\rho}{\hookrightarrow}_{,\Sigma} k :: -; s_2 \blacktriangleright s_1 \otimes \mu_0$。(1) 根据逆向分析，$\vdash^R_\Sigma k \blacktriangleleft: (\Psi_2, R_1) \rightsquigarrow (\Psi' \rho)$，且对所有 $\rho' \in R_1$，我们有 $\cdot; \Psi \vdash^R_\Sigma s_1 @ \rho' \dashv \Psi_1$ 和 $\cdot; \Psi_1 \vdash^R_\Sigma s_2 @ \rho' \dashv \Psi_2$。根据 KS-Seq，$\vdash^R_\Sigma k :: -; s_2 \blacktriangleleft: (\Psi_1, R_1) \rightsquigarrow (\Psi' \rho)$。根据 KS-PopStmt，$\Psi \vdash^R_\Sigma k :: -; s_2 \blacktriangleright s_1 @ \ \rho \dashv \Psi'$。(2) 未添加同步边，因此良构性得以保留。

(3) 不会向 W 或 L 添加任何项。(4) L, W 不变。(5) 不会更改任何签名。(6) 不会向 W 添加任何项。(7) 不会向 W 添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 若 $L(\beta) = (a, \_, \_)$，则对于某个 k^p，有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。此情况保持不变。• Seq2。接着 $\mu = a \overset{\rho}{\hookrightarrow}_{,\Sigma} k :: -; s_2 \blacktriangleleft \otimes \mu_0$ 和 $\mu' = a \overset{\rho}{\hookrightarrow}_{,\Sigma} k \blacktriangleright s_2 \otimes \mu$。(1) 根据逆向分析，$\vdash^R_\Sigma k \blacktriangleleft : (\Psi'' R_1) \rightsquigarrow (\Psi' \rho)$，并且对于所有 $\rho' \in R_1$，有 $\cdot; \Psi \vdash^R_\Sigma s_2 @ \rho' \dashv \Psi''$。根据 KS-PopStmt，$\Psi \vdash^R_\Sigma k \blacktriangleright s_2 @ \rho \dashv \Psi'$。(2) 没有添加同步边，因此良构性得以保持。(3) 不会向 W 或 L 添加任何项。(4) L, W 不变。(5) 不会更改任何签名。(6) 不会向 W 添加任何项。(7) 不会向 W 添加任何项。(8) 此步骤不会拆分或重新分配权限。(9) 若 $L(\beta) = (a, \_, \_)$，则对于某个 k^p，有 $k \underset{\beta,a,\rho}{\rightsquigarrow} k^p$。此情况保持不变。

□

接下来，我们证明一个进展结果：如果某个配置满足这些不变式，那么每个未阻塞的线程都可以按照动态语义执行一步。

定理 5.4。假设 $\mu; \sigma; W; L \mid g$ 满足定义 *5.2* 中的不变式。那么对于所有 $a \overset{\rho}{\hookrightarrow}_{,\Sigma} K \in \mu$，要么 $K = \epsilon\blacktriangleleft$，要么 $a \overset{\rho}{\hookrightarrow}_{,\Sigma} K \otimes \mu_0; \sigma; W; L \mid g \Rightarrow \mu\prime; \sigma\prime; W\prime; L\prime \mid g\prime$，其中 $a \overset{\rho}{\hookrightarrow}_{,\Sigma} K \otimes \mu_0 = \mu$。

证明。根据不变式以及对 Global 的逆向分析，我们有 $\Psi \vdash^R_\Sigma K @ \rho \dashv \Psi\prime$。对该推导进行归纳。完整证明见补充材料。

• KS-PopInstr。然后 $K = k \triangleright i$ 和 $\vdash^R_\Sigma k \blacktriangleleft : (\tau, \Psi\prime\prime R_1) \rightsquigarrow (\Psi\prime \rho)$ 以及 $\cdot; \Psi \vdash^R_\Sigma i \sim : \tau @ \rho \dashv \Psi\prime\prime$。对后一个推导进行嵌套归纳。– 生成。应用生成。– 新建引用。应用新建引用。– 解引用。然后，由于上下文为空，根据对 RefVal 的逆向分析，$i = !\mathrm{ref}[s]$ 和 $s \sim \tau \in \Sigma$。根据内存类型，$s \in dom(\sigma)$。应用解引用。– 更新。应用更新。– 等待。然后，由于上下文为空，根据对条件变量的逆向分析，$i = \mathrm{wait\ cv}[\alpha]$。应用等待。– 信号。然后，由于上下文为空，根据对条件变量的逆向分析，$i = \mathrm{signal\ cv}[\alpha]$。根据 $W(\alpha)$ 是否为空，应用 Signal1 或 Signal2。– 提升。然后，由于上下文为空，根据对条件变量的逆向分析，$i = \mathrm{promote\ cv}[\alpha]\ \mathrm{to}\ \rho_2$。应用提升。– 新建条件变量。应用 NewCV。– 新建互斥锁。应用 NewMutex。

带同步的响应式并行

- KS-PopStmt。然后 $K = k \blacktriangleright s$和 $\vdash_\Sigma^R k \blacktriangleleft$: （$\Psi''R_1$） $\rightsquigarrow$ （$\Psi'\rho$）以及 $\cdot$; $\Psi \vdash_\Sigma^R s@\rho \dashv \Psi''$。对后一个推导进行嵌套归纳。– WithLock。然后，由于上下文为空，根据对互斥锁的逆向分析， $s =$（with mutex[β] s'）。如果 L（β） $= \bot$，则应用 WithLockS1。否则，如果 L（β） $=$（$a'u'1$， $u'2$）并且 $\rho \preceq$ Prio$g$（a'），则应用 WithLockS2。否则，根据不变量 5，存在 ρ_β，使得 $\beta@\rho_\beta \in \Sigma$。我们必须有 Prio$g$（$a'$）$\neq \rho_\beta$，因为 $\rho \preceq \rho_\beta$ 和 $\rho \not\preceq$Prio$g$（a'）。因此，根据这些不变式，在 μ_0或 W 中存在一个 $b \underset{\rho\ \Sigma}{\hookrightarrow}{}'\ {}'K_b$，使得 $K_b \underset{\beta,b,\rho}{\rightsquigarrow}{}'K''$，并且我们可以分别应用 WithLockS3 或 WithLockS4。– Let。应用 Let1。– If。然后 $s =$（if v then s_1 else s_2）。由于上下文为空，根据对 natI 的逆向分析，$v = n$ 对于某个 n。应用 If1 或 If2。– While。应用 While。– Skip。应用 Skip。– Seq。应用 Seq1。

–

- KS-PushInstr。然后 $K = k \lhd v$和 $\vdash_\Sigma^R k \lhd$: (τ， Ψ， R_1) $\rightsquigarrow$($\Psi'\rho$) 和 $\cdot \vdash_\Sigma^R v$: τ。通过逆向分析， $k = k_0$ :: let $x =$– in s。应用 Let2。
- KS-PushStmt。然后 $K = k\blacktriangleleft$和 $\vdash_\Sigma^R k \blacktriangleleft$: ($\Psi$， R_1) $\rightsquigarrow$($\Psi'\rho$)。对该推导进行嵌套归纳。– KS-Empty。然后 $K = \epsilon\blacktriangleleft$。– KS-Seq。应用 Seq2。– KS-WithLockS。然后，由于上下文为空，通过对 Mutex 进行逆向分析， $k = k_0$ :: with* mutex[β] –。根据 $W(\beta)$ 是否为空，应用 WithLockE1 或 WithLockE2。– KS-WithLockSP。然后，由于上下文为空，通过对 Mutex 进行逆向分析， $k = k_0$ :: with*[a'; ρ'] mutex[β]

–.

根据 $W(\beta)$ 是否为空，应用 WithLockE3 或 WithLockE4。

□

根据进展和保持性可直接得出，任何可从初始配置到达的配置都能继续取得进展（或已执行完毕），并且包含一个良构图。

定理 *5.5*（可靠性）。假设 $\Psi \vdash^R \cdot K@\ \rho \dashv \Psi'$，其中 Ψ和 Ψ'是有效的权限映射，且 $a \underset{\rho}{\hookrightarrow}{}_{,\cdot}$ $K\ \emptyset\ \emptyset\ \emptyset \mid \emptyset \Rightarrow^* \mu\ \ \sigma\ \ W\ \ L\ \mid g$ g $a \underset{\rho\ \Sigma}{\hookrightarrow} K \in \mu$ $K = \epsilon\blacktriangleleft$ $a \underset{\rho\ \Sigma}{\hookrightarrow} K \otimes ; ; ; ';$ $\mu\ \ \sigma\ \ W\ \ L\ \mid g \Rightarrow \mu\ \ \sigma\ \ W\ \ L\ \mid g$ μ $'; '; ' '$。那么 $'$ 是良构的，并且对于所有 , , 要么 或 , $'0; '; '; ' ' ''; ''; ''; '' ''$，其中 $' = a \underset{\rho}{\hookrightarrow}{}_{,\Sigma}$ $K \otimes \mu$ $'0$。

证明。根据 Global，初始配置是良类型的。其他所有不变式均由初始条件满足。根据定理 5.3 和 5.4 的归纳应用，结论成立。□

5.3 扩展

下面我们将讨论两种目前尚未在 λ_s^4 中建模、但可以轻松添加的同步操作。迄今为止，我们没有纳入这些操作，是为了尽可能简化语义和证明，并聚焦于关键要点。

trylock。在许多基于互斥锁的同步实现中，trylock 是一种非阻塞结构，它会尝试获取互斥锁；如果互斥锁已被锁定，trylock 会立即返回，并通过返回值或错误代码表明获取互斥锁失败。我们可以为 with v s 建模一个类似的变体；

我们可以称之为 trywith。在我们的语法中，如果失败，trywith 会执行一条替代语句，或将指定变量设为表示失败的值。trywith 的使用限制与 with 完全相同。其动态语义将包含一条与 WithLockS1 对应的规则，以及一条附加规则，用于在 $L(\beta) \neq \bot$ 时执行所需的替代操作。

广播。信号操作会唤醒一个正在等待条件变量的线程。许多条件变量的实现也包含广播原语，可唤醒所有等待线程。将此操作纳入 λ_s^4会很直接；其类型限制与信号操作的类型限制相同。动态语义规则（类似于 Signal1）会将 $W(\alpha)$中的所有线程重新加入线程池，并从当前线程向所有等待线程添加同步边。

6 实现与案例研究

本节介绍我们对类型系统规则的 Rust 和 C++ 实现，以及为评估这些规则的易用性而开展的相应案例研究。两种实现都提供了一个库，支持以不同优先级生成线程并执行同步，也都通过静态检查来强制执行第 2 节中讨论的同步原语限制的不同版本。我们使用三个应用基准开展了案例研究，其中两个是从头开发的，另一个是现实世界中的交互式应用，即 Memcached 对象缓存服务器 [Mem 2009] (v1.5.13)；我们将它移植为使用我们的 C++ 类型系统。我们先讨论 Rust 实现，再讨论 C++ 实现，重点介绍它与 Rust 实现的不同之处，最后介绍案例研究，说明我们在使用类型系统时的经验。

6.1 Rust 实现

Rust 实现着重利用 Rust 丰富的类型系统（其中包含所有权和亲和性的概念）来限制条件变量的使用；它不会强制执行对互斥锁的限制。我们的库为线程处理和条件变量操作提供封装器，并在调用 Rust 标准线程库中相应的操作之前，检查优先级和所有权方面的相关限制。整个库共包含 551 行代码。[7]

为编码第 2 节中的七项限制，我们的库需要表示优先级和所有权。优先级表示为实现 `Priority` 特征的类型。优先级只是标记，永远不会被实例化；它们仅用作其他泛型构造的类型参数。另一个特征 `Ge`（大于或等于）用于建立这些类型在类型层面的偏序关系。线程的优先级由一个以优先级为参数的 `Token` 类型表示。每个线程都有一个令牌；某些库函数需要传入令牌的引用，以此证明当前线程的优先级。`Token` 类型的定义确保安全代码无法访问其他线程的令牌，并且只能通过生成一个线程来创建与之配套的新令牌。

该库还通过 `Owned` trait 定义了对应于各个所有权级别的类型（独占、共享 和 无访问权限）。这些所有权级别不应与 Rust 类型系统本身的所有权特性（它们与之相似）混淆；这些类型会在条件变量的类型中明确表示所有权。每个类型为条件变量的条件变量，都由其优先级以及各优先级对应的其中一种所有权级别进行参数化。这带来了一个限制：优先级的数量必须由库预先固定；在我们的示例中，三个级别（低、中 和 高）就足够了。我们定义了一个拆分 方法，利用 Rust 默认仿射类型系统，确保三个所有权优先级各自遵循适当的规则（例如，独占 状态意味着没有其他指向该条件变量的“句柄”拥有独占 或共享 访问权限

[7]代码行数由 tokei 统计。

带同步的响应式并行

```
1 let cv = Condvar::new(token).提升::<高>(); 2 let (cv_to_p, cv_to_c) =cv.hollow_拆分(); 3 l
et c_th =prio::生成_and_传递_hollow(cv_to_c, |cv, token: &Token<高>|4 {cv.dummy_等待
(token);}); 5 let (cv2, p_th) = prio::生成_and_拆分(cv_to_p,token, |cv, token: &Token<高>|6
{cv.notify_one(token);}); 错误[E0277]: 未满足 trait bound `无访问权限: Partial`
```

图 22. Rust 中类型不正确的生产者-消费者示例的代码框架，以及由此产生的错误消息。

（相同优先级）。Rust 的仿射类型系统还确保这些操作返回的条件变量句柄会在后续使用，而其他句柄（例如已拆分出的旧句柄）则会失效。条件变量的优先级和所有权使我们能够对条件变量的使用施加适当限制。施加这些限制的一个重要方面是，必须限制在不同线程之间共享条件变量。我们通过另一种 traitPrioSend以及多个函数来实现这一点，这些函数允许以安全的方式在线程之间传递条件变量。

例如，图 22 展示了图 3 中类型不正确的生产者-消费者示例的简化 Rust 代码（仅展示主函数体；生产者和消费者线程的代码已删减，只保留信号和等待操作）。主线程使用类型为token<低>的令牌token 进行初始化。第 1 行中，我们使用此令牌创建一个条件变量，并立即将其提升至高优先级。Rust 能够推断出cv的类型为条件变量<高, 无访问权限, 无访问权限, 独占>，这表示该条件变量具有高优先级，并且由高优先级独占，在低优先级或中优先级下均无所有权，符合限制 4。第 2 行中，我们将条件变量拆分为两个句柄，以便传递给两个线程。hollow_split 方法会生成其中一个句柄cv_to_c，它不具有所有权。接着，我们将此条件变量传递给消费者线程（根据其令牌类型可知，该线程具有高优先级），由它在该条件变量上等待。另一个条件变量句柄则在第 5 至 6 行传递给生产者线程，该线程同样具有高优先级，并使用notify_one 方法发送信号。根据限制 3，这次生成线程的操作应当类型不正确，因为主线程在低优先级下没有所有权；事实上，编译代码时，第 5 行会触发图底部所示的错误。

差异与局限。我们的 Rust 实现与第3节讨论的类型系统在一些细节上有所不同。首先，Rust 库支持的优先级数量固定。此外，条件变量的类型明确体现了所有权（尽管 Rust 能够自动推断其中一些类型），程序员必须手动调用方法来适当地拆分和转移所有权。最后，可以使用 不安全的 代码绕过用于强制执行库某些限制的线程安全机制。

6.2 C++实现

我们还实现了一个 C++ 库（由 1,252 行代码组成），用于近似实现 λ_s^4 的功能和限制；除了 Rust 库的功能外，C++ 实现还执行了我们对互斥锁的限制。C++ 库定义了包装器，用于封装 I-Cilk [Muller 等人（2020）；Singer 等人 2020]提供的线程处理和同步功能。C++ 库采用源自先前工作的策略来表示优先级 [Muller 等人 2020]：每个优先级都表示为一个 类，两个优先级之间的关系则通过继承的类层次结构来表示。每个线程都有一个在编译时已知的优先级，该优先级通过以适当的优先级类型调用参数化函数并生成线程来初始化。类似地，每个条件变量也使用其自身的优先级类型初始化，每个互斥锁则使用其上限优先级类型初始化。当线程操作互斥锁或条件变量时，类型系统会根据第 2 节讨论的限制，使用静态断言和 是_基类_的 来检查优先级反转。

线程和互斥锁或条件变量的优先级类型。由于每个临界区都需要根据当前线程的优先级和互斥锁的优先级上限进行类型检查，因此我们要求程序员将每个临界区提升为独立的参数化函数，以便显式地用这两个优先级进行实例化。

将所有权这一概念编码到类型系统中更加复杂，因为 C++ 采用非仿射类型系统，并不天然支持静态所有权检查。所有权以枚举表示（无、共享或独占）。创建条件变量时，会为其指定一个优先级和一个所有权状态列表，其中每个优先级级别对应程序中使用的一个优先级。条件变量通过可变参数模板实现，使不同程序能够以不同的最大优先级初始化条件变量。因此，与 Rust 库不同，优先级的数量并非由库固定（不过仍必须在编译时确定）。这种模板化的条件变量充当包装器，下文称为具有相应优先级和所有权类型的包装器_条件变量。

与 Rust 一样，我们在包装器_条件变量上定义了显式的所有权变更操作，例如对条件变量执行所有权拆分、所有权转移和提升；此类操作会使表示先前所有权级别的旧包装器_条件变量失效。这是 C++ 实现面临的一项重大挑战，因为它不像 Rust 那样会被自动强制执行。所有权类型的变更是通过在当前线程 *a* 的词法作用域中静态地使原包装器_条件变量 变量失效，并创建一个与同一底层条件变量关联的新包装器_条件变量 变量来实现的；该变量将在 spawn语句之后使用。为了使原包装器_条件变量 变量失效，我们使用了一个计数器 编译时计数器，该计数器可以在编译时递增。该计数器 提供了一个next 递增方法，它在编译时求值为一个整数字面量；每当当前翻译单元中出现next 时，该值就会加一。

我们使用这些编译时计数器作为“毒化标志”，以指示给定包装器_条件变量 变量的有效性。每当引入新的包装器_条件变量 变量时（无论是由程序员显式声明，还是通过所有权变更宏引入），都会在同一词法作用域中引入一个与该包装器_条件变量 关联的 counter 类型局部变量。计数器初始值为 0，表示包装器_条件变量有效。当调用任何会使包装器_条件变量失效的所有权变更宏时，该宏也会对计数器调用递增，使其值变为 1，表示包装器_条件变量已失效。对包装器_条件变量执行的任何操作——发信号或调用所有权变更宏——都会使用静态断言来确保包装器_条件变量 仍然有效，并执行第 2 节中讨论的必要检查。

差异与局限。与 Rust 实现之间的许多差异（例如，显式更改所有权）在 C 实现中也同样存在。此外，在 C 中，if 语句可以更改所有权映射，前提是条件分支和 else 分支以相同方式更改映射（第 3 节图 10 中的 If）；而在 C 中，如果分支需要更改所有权，程序员必须在if 语句之前显式调用所有权变更宏。这是因为我们没有可靠的方法来检查两个分支是否以相同方式更改所有权映射。

为实施类型规则，我们提供了额外的功能，并施加了某些编程限制。首先，为确保传递条件变量时同时转移所有权，对包装器_条件变量 调用的操作会进行名称修饰。因此，程序员必须使用相应的宏，在生成带有条件变量的线程时转移所有权，并在函数入口处调用一个宏，以启用条件变量的使用。缺少其中任何一项操作，代码都将无法编译。其次，为了允许在循环中拆分条件变量，库提供了另一种拆分宏，该宏不会使输入的包装器_条件变量 失效，并会断言拆分不会改变输入的所有权。最后，为确保 if-else 和循环结构能恰当地与条件变量交互，库要求程序员使用提供的专用结构。这些专用结构确保不存在作用域超出该结构主体范围的条件变量。

带同步的响应式并行

会在这些构造中失效，也就是说所有权权限级别不会发生变化。库宏会将普通 C++ if / 循环构造定义为空，使得编译器在使用条件变量的翻译单元中遇到普通 C++ 构造时输出错误。最后，与先前工作 [Muller 等人 2020], 一样，程序员不应使用不安全的类型转换来修改优先级或所有权。

6.3 应用案例研究

为评估我们类型系统的易用性，我们用 Rust 开发了一个聊天服务器（356 行代码），用 C++ 开发了一个电子邮件客户端（1,252 行代码），并将 Memcached 服务器这一大型交互式应用（20,100 行 C 代码）移植为使用我们的 C++ 类型系统。我们依次介绍各个应用，并讨论相关经验。

Rust 聊天服务器。这个简单的多用户聊天服务器采用分层线程集。一个优先级的线程负责处理每位已连接用户的 TCP 流，并接受和处理新用户。较低优先级的线程则将近期消息写入频道元数据。每个频道都关联一个条件变量；任何已连接客户端发送新消息时，都会发出信号。每位已连接用户都会生成一个新线程，该线程接收 TCP 流和条件变量。这些线程既是生产者也是消费者：既发送已连接用户的消息，也会定期等待条件变量，以接收其他已连接客户端发来的新消息。

C 语言中的电子邮件客户端++. 利用我们的 C++ 实现，我们基于先前的工作 [Mulleret al. 2020] 实现了一个多用户共享电子邮件服务器。该服务器使用 5 个优先级级别。在最高优先级下，服务器接受新的客户端连接（每个连接对应一个线程），侦听已连接客户端的请求，并生成其他类型的线程来处理收到的请求：发送线程处理发送电子邮件的请求（最高优先级）；排序 线程处理对电子邮件进行排序的请求（第二高优先级）；压缩 线程压缩电子邮件，打印 线程处理打印电子邮件的请求（第三高优先级）。此外，还有一个 出队 线程，负责将发送电子邮件时产生的压缩任务出队，并生成线程来执行压缩（第四高优先级）。负责执行设置和拆卸任务的主线程优先级最低。

有趣的交互之处在于如何生成压缩线程。在多生产者/单消费者模型中，两种不同的线程相互交互：(多个)发送线程充当生产者，而出队线程充当消费者。由于发送线程的数量在编译时未知（取决于发送请求的数量），我们在循环中使用替代拆分宏（不会使输入失效），在生成每个发送线程时拆分出 CV 信号权限。当发送线程将收件箱中的邮件数量增加到超过未压缩邮件的阈值时，它会将要压缩的信息加入队列，并向出队线程发送信号，使其唤醒并生成一个压缩线程。

C语言中的 Memcached 对象缓存服务器。 Memcached 对象缓存服务器 [Mem 2009] 是一种分布式内存缓存。它是一种键值存储，核心用途是维护一个对象哈希表，客户端可以更新或检索其中的对象。Memcached 使用 C 语言编写，并通过 pthreads 和 I/O 多路复用来处理大量客户端。

Memcached 的主线程执行大部分设置代码，并生成其他长期运行的线程，例如在必要时调整哈希表大小的调整线程、维护哈希表各存储桶中项目（大致）最近最少使用（LRU）顺序的LRU 维护线程、释放缓存中已经变得“冷”的项目的 LRU 爬虫线程、用于在 Memcached 内部内存池的不同大小类别之间平衡空闲内存的slab维护线程，以及聚合并记录来自其他线程的消息和统计信息的日志线程。

我们将应用程序划分为三个不同的优先级级别——每当实际执行调整大小时，它都会作为高优先级任务执行（由负责调整大小的线程生成）；另一方面，任何与日志记录相关的活动都作为低优先级任务执行。处理客户端请求、维护缓存的 LRU 以及管理内存池相关的活动，则作为中优先级任务执行。这样做很方便，因为这些任务会在公共锁的临界区内对条件变量发信号并等待。[8] 我们选择保留单一的中优先级，因为此优先级下的所有任务彼此依赖（例如，如果 slab 线程以较低优先级运行，或者 LRU 爬虫线程未能及时释放内存，客户端请求可能会耗尽内存）。如果我们希望采用更细粒度的优先级级别，就可以更改某些临界区的优先级，并在更多地方将条件变量从较低优先级提升到较高优先级。

移植 Memcached 服务器需要修改约 11,400 行 memcached 代码，约占 57%。这项工作由一名研究生完成，他同时还负责设计和维护类型系统的 C++ 实现。若将适配和扩展 C++ 库实现所花费的时间也计算在内，此次转换约需 80 人时。转换涉及的许多任务都是机械性修改，包括将代码从 C 转换为 C++，以及按照我们的库的要求进行的机械性修改（例如将函数转换为命令、转换 if 语句）；借助重构工具，这些工作可以显著提速。我们估计，不使用重构工具，转换本身所需时间不到 40 人时；使用重构工具则约需 8 至 16 人时。

讨论。 能够使用我们的 C++ 类型系统对 Memcached 服务器进行类型检查，让我们对类型规则不会过于严格有了一些信心。在很大程度上，从头开发应用程序（即聊天服务器和电子邮件客户端）时，将优先级注释纳入其中相当直接。由于类型处理不当而遇到的编译错误（例如，在条件变量所有权转移时，忘记在必要的代码位置使用正确的拆分函数）很容易修复，而且确实有助于确定线程的正确优先级。例如，Rust 聊天服务器的初始设计中，处理已连接用户的线程优先级高于处理新用户的线程，但这一设计存在一个细微的优先级反转问题，静态限制发现了该问题。

将 Memcached 服务器转换过来的过程不那么直接，因为其代码库相当成熟，并采用了一些与我们编码类型规则的方式不太契合的编程模式。例如，获取锁和释放锁的操作并不总是严格嵌套的（例如，在一个函数中获取锁，却在另一个函数中释放锁），因此我们需要重构代码，以便将临界区提取为单独的参数化函数。另一个例子是，在 if-else 和循环等控制结构中使用条件变量。虽然将代码改为使用针对这些结构的专用宏，过程相当机械且直接，但如果这类结构存在多层嵌套（Memcached 中就有这种情况），便容易出错。如果从头编写代码，我们会重写控制流，以避免控制结构复杂嵌套。

这些实现和案例研究也证明，我们的系统实用且并不过于复杂。所有实现者都不是类型系统理论方面的专家。Rust 库由一名高年级本科生实现，他没有参与类型系统的设计，在阅读第 2 节的介绍并与第一作者进行一两次讨论后完成了实现。Rust 聊天服务器则由另一名高年级本科生实现，他没有参与类型系统的设计或实现，并接受了类似的准备。

[8]如第 2 节所述，为避免优先级反转，获取同一把锁的线程必须具有相同的优先级。

7 相关工作

协作式与竞争式线程处理。关于协作式线程处理的研究可以追溯到 20 世纪70年代末和80年代。当时，Id [Arvind 和 Gostelow1978] 以及 Multilisp [Halstead 1985, 1984]等系统采用了这种方式，即并行工作者协作完成一项（主要是计算性的）任务。此后，人们在许多语言的背景下研究了这一模型，包括 ML 的并行方言 [Arora et al. 2021; Fluet et al. 2011, 2007; Raghunathan et al. 2016], Haskell [Chakravarty et al. 2007; Peyton Jones et al. 2008], 以及 Java [Charles et al. 2005; Imam and Sarkar 2014]。竞争式线程处理，或称并发，是指利用线程提升程序的响应性、延迟表现或可组合性；几十年来，这一直是研究与实践的主题，随之而来的调度和同步问题也是如此。该领域的相关研究成果浩如烟海，无法在此详尽回顾，但我们向有兴趣的读者推荐 Silberschatz 等人撰写的精彩综述 [2005]。

部分作者此前的工作 [Muller 等人 2017, 2018] 指出，为协作式线程处理添加可证明的响应性保证，需要加入用于提升交互式线程优先级的语言构造，并进行谨慎控制以避免优先级反转（本节稍后将讨论）。此后，这项工作扩展到处理公平性 [Muller 等人 2019] 以及可变状态 [Muller 等人 2020]。

高效调度与代价语义。Brent [1974], 以及后来的 Eager 等人 [1989], 根据并行程序的工作量 和跨度 对调度长度进行了界定。代价语义 [Rosendahl 1989; Sands 1990] 将传统操作语义扩展为跟踪资源使用情况。自 NESL 语言早期的工作 [Blelloch and Greiner 1995, 1996]以来，研究者便使用基于图或有向无环图表示的并行程序代价语义。

本文的代价语义直接建立在我们近期提出的可变状态并行程序模型之上 [Muller 等人 2020]。该工作引入了弱边，用于表示由内存读写所导致的先发生依赖；还引入了强化，以便在计算工作量和跨度时将弱边转换为更标准的依赖边。基于这些定义，我们证明了一个具有 Brent 风格的调度界（涵盖交互式程序的吞吐量和响应时间），并假设图是良构的，也就是说不存在优先级反转；该条件通过静态方式强制保证。本文使用弱边表示由互斥锁竞争所导致的先发生关系，并结合静态和动态技术来避免优先级反转。我们的可靠性结果以这些定理为基础，并从多个方面对其进行了推广。此外，我们的扩展还对运行时优先级变化进行建模（以实现优先级天花板协议），而此前关于响应式并行性的工作并未处理这一问题。

优先级反转。自20世纪80年代以来，人们一直在多种语言和系统中研究优先级反转（例如，[Cornhill 和 Sha，1987年；Lampson 和 Redell 1980]），并提出了多种解决方案，包括优先级继承 和优先级上限 [Sha 等人 1990]提出的方法；我们实现后者，因为与优先级继承相比，它通常能减少优先级提升次数，也更容易推理，不过这要求预先知道每个互斥锁的优先级上限。

使用条件变量时的优先级反转控制问题研究得少得多。如第 1 节和第 2 节所述，标准的动态优先级继承技术不足以避免使用条件变量的程序中出现优先级反转。Cucinotta [2013], 注意到了这一问题，并提出允许程序员指定任务之间的这类依赖关系，以便运行时调度器在必要时执行优先级继承。我们使用类型系统自动发现这些依赖关系，而不要求程序员指定，并在编译时排除由此产生的优先级反转。据我们所知，本文提出了首个用于避免使用互斥锁和条件变量的程序中出现优先级反转的类型系统。

Babaoğlu 等人 [1993] 提出了一种优雅的基于图的优先级反转形式化方法及避免此类反转的技术，这在某些方面与本文使用的基于图的模型相似。然而，他们的表述基于线程在执行过程中彼此阻塞时的快照，不足以表示涉及线程生成历史的复杂优先级反转，例如图 3 中代码所引发的反转。

所有权与权限。我们针对条件变量的类型限制严重依赖于所有权这一概念。所有权已被应用于许多系统中，通常用于防止并发编程中的数据竞争，或实现安全的内存管理。用于追踪所有权及相关属性的机制层出不穷，其中大多源自线性逻辑 [Girard 1987年；Reynolds 1974]。这些机制包括能力（例如， [Ahmed 等人 2007年；Crary 等人 1999]）和别名类型 [Smith 等人 2000]，，并已被纳入诸如 Cyclone [Grossman 等人 2002] 和 Rust 之类的语言中（也因此启发了我们的 Rust 实现）。我们最受分数权限的启发 [Boyland 2003]，，不过我们采用了分数权限的一种相当常见的放宽形式（例如， [Heule 等人 2011年；Naden 等人 2012]），其限制性较低。

在语言 L^3 中，Ahmed 等人 [2007] 在类型系统中使用了能力来支持强更新，强更新会改变内存单元的类型（强更新需要所有权，因为改变类型会使对该单元的其他引用失效）。我们的 promote 操作可视为一种强更新，因为它会改变条件变量句柄的优先级，进而改变其类型；而提升所受的限制则大致基于 L^3 中的限制。然而，提升与强更新不同，它不会使指向该条件变量的所有其他句柄失效；其他线程仍可使用句柄，以更高的优先级发送信号，或以更低的优先级等待。我们通过分数权限允许这样做，而 L^3 中未考虑分数权限。

8 结论

本文介绍了 λ_s^4，这是一种响应式并行编程演算，可通过互斥锁和条件变量实现同步。λ_s^4 的类型系统能够静态阻止使用条件变量时可能出现的优先级反转，并保证 λ_s^4 的代价语义所生成的图满足强调度界限，从而排除优先级反转。该类型系统可以近似编码到 Rust 和 C++ 中，这些编码可用于编写规模可观的程序。未来一个令人振奋的研究方向是开发编译器，直接支持这些类型限制；这样的定制编译器可以在减少甚至无需程序员干预的情况下跟踪条件变量的所有权，并通过类型推断自动确定条件变量的优先级和互斥锁的优先级上限。

参考文献

2009年。Memcached。https://memcached.org/。访问于2019年7月。Amal Ahmed、Matthew Fluet 和 Greg Morrisett。2007年。L^3: 一种带位置的线性语言。*Fundam. Inform.* 77, 4 (2007), 397–449。Jatin Arora、Sam Westrick 和 Umut A. Acar。2021年。可证明空间高效的并行函数式编程。5, POPL，文章 18（2021年1月），33页。https://doi.org/10.1145/3434299 Arvind 和 K. P. Gostelow。1978年。*Id* 报告：一种异步语言与计算机。技术报告 TR-114。加利福尼亚大学欧文分校计算机与信息科学系。Özalp Babaoğlu、Keith Marzullo 和 Fred B. Schneider。1993年。优先级反转的形式化。实时系统 5, 4 (1993), 285–303。Guy Blelloch 和 John Greiner。1995年。顺序函数式语言中的并行性。载于第七届函数式编程语言与计算机体系结构国际会议论文集（*FPCA '95*）。ACM，226–237。Guy E. Blelloch 和 John Greiner。1996年。NESL 的一种可证明具有时间与空间效率的实现。载于第一届 *ACM SIGPLAN* 函数式编程国际会议论文集。ACM，213–225。John Boyland。2003年。使用分数权限检查干扰。载于国际静态分析研讨会。Springer，55–72。Richard P. Brent。1974年。一般算术表达式的并行求值。*J. ACM* 21, 2 (1974), 201–206。

带同步的响应式并行

Manuel M. T. Chakravarty、Roman Leshchinskiy、Simon L. Peyton Jones、Gabriele Keller 和 Simon Marlow。2007年。数据并行 Haskell：现状报告。载于 2007 年 POPL 声明式多核编程专题研讨会论文集，DAMP 2007，法国尼斯，2007年1月16日。10–18。Philippe Charles、Christian Grothoff、Vijay Saraswat、Christopher Donawa、Allan Kielstra、Kemal Ebcioglu、Christoph von Praun 和 Vivek Sarkar。2005年。X10：一种面向非均匀集群计算的面向对象方法。载于第二十届 ACM SIGPLAN 面向对象编程、系统、语言与应用年会论文集（美国加利福尼亚州圣迭戈）（OOPSLA '05）。ACM，519–538。Dennis Cornhill 和 Lui Sha。1987年。Ada 中的优先级反转。Ada Letters VII, 7（1987年11月），30–32。https://doi.org/10.1145/36072.36073 Karl Crary、David Walker 和 Greg Morrisett。1999年。能力演算中的类型化内存管理。载于第 26 届 ACM 编程语言原理专题研讨会会议录，美国德克萨斯州圣安东尼奥。262–275。Tommaso Cucinotta。2013年。条件变量上的优先级继承。第九届嵌入式实时应用操作系统平台国际研讨会论文集（OSPERT 2013）。https://www.researchgate.net/publication/242330394_Priority_Inheritance_on_Condition_Variables Derek L. Eager、John Zahorjan 和 Edward D. Lazowska。1989年。并行系统中的加速比与效率。IEEE Transactions on Computing 38, 3 (1989), 408–423。Matthew Fluet、Mike Rainey、John Reppy 和 Adam Shaw。2011年。Manticore 中的隐式线程并行性。Journal of Functional Programming 20, 5-6 (2011), 1–40。Matthew Fluet、Mike Rainey、John Reppy、Adam Shaw 和 Yingqi Xiao。2007年。Manticore：一种异构并行语言。载于 2007 年声明式多核编程专题研讨会论文集（法国尼斯）（DAMP '07）。37–44。Jean-Yves Girard。1987年。线性逻辑。Theoretical Computer Science 50, 1 (1987), 1–101。https://doi.org/10.1016/0304-3975(87)90045-4 Dan Grossman、Greg Morrisett、Trevor Jim、Michael Hicks、Yanling Wang 和 James Cheney。2002年。Cyclone 中基于区域的内存管理。载于 ACM SIGPLAN 2002 年编程语言设计与实现会议论文集（德国柏林）（PLDI '02）。计算机协会，纽约，美国，282–293。https://doi.org/10.1145/512529.512563 Robert H. Halstead。1985年。MULTILISP：一种用于并发符号计算的语言。ACM Transactions on Programming Languages and Systems 7 (1985), 501–538。Robert H. Halstead, Jr. 1984年。Multilisp 的实现：多处理器上的 Lisp。载于 1984 年 ACM LISP 与函数式编程专题研讨会论文集（美国得克萨斯州奥斯汀）（LFP '84）。ACM，9–17。Per Brinch Hansen。1973年。操作系统原理。Prentice-Hall, Inc. Per Brinch Hansen。1975年。并发 Pascal 编程语言。IEEE Transactions on Software Engineering 2 (1975), 199–207。Stefan Heule、Rustan Leino、Peter Müller 和 Alexander J. Summers。2011年。无需分数的分数权限。载于 FTfJP'11，2011年7月26日，英国兰开斯特。Charles Antony Richard Hoare。1974年。监视器：一种操作系统结构化概念。载于《并发编程的起源》。Springer，272–294。Shams Mahmood Imam 和 Vivek Sarkar。2014年。Habanero-Java 库：面向多核编程的 Java 8 框架。载于 2014 年 Java 平台虚拟机、语言与工具编程原理与实践国际会议，PPPJ '14。75–86。Butler W. Lampson 和 David D. Redell。1980年。Mesa 中进程与监视器的实践经验。Commun. ACM 23, 2 (1980), 105–117。Stefan K. Muller、Umut A. Acar 和 Robert Harper。2017年。响应式并行计算：连接竞争式与协作式线程处理。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（西班牙巴塞罗那）（PLDI 2017）。ACM，纽约，美国，677–692。Stefan K. Muller、Umut A. Acar 和 Robert Harper。2018年。竞争式并行性：正确设置优先级。载于第 14 届 ACM SIGPLAN 函数式编程国际会议论文集（ICFP '18）。Stefan K. Muller、Kyle Singer、Noah Goldstein、Umut A. Acar、Kunal Agrawal 和 I-Ting Angelina Lee。2020年。基于 future 与状态的响应式并行性。载于第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国伦敦）（PLDI 2020）。计算机协会，纽约，美国，577–591。https://doi.org/10.1145/3385412.3386013 Stefan K. Muller、Sam Westrick 和 Umut A. Acar。2019年。响应式并行性中的公平性。载于第 24 届 ACM SIGPLAN 函数式编程国际会议论文集（ICFP 2019）。Karl Naden、Robert Bocchino、Jonathan Aldrich 和 Kevin Bierhoff。2012年。借用权限的类型系统。载于第 39 届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会论文集（美国宾夕法尼亚州费城）（POPL '12）。计算机协会，纽约，美国，557–570。https://doi.org/10.1145/2103656.2103722 Simon L. Peyton Jones、Roman Leshchinskiy、Gabriele Keller 和 Manuel M. T. Chakravarty。2008年。驾驭多核：Haskell 中的嵌套数据并行性。载于 FSTTCS。383–414。Ram Raghunathan、Stefan K. Muller、Umut A. Acar 和 Guy Blelloch。2016年。并行程序的分层内存管理。载于第 21 届 ACM SIGPLAN 函数式编程国际会议论文集（日本奈良）（ICFP 2016）。ACM，纽约，美国，392–406。John C. Reynolds。1974年。迈向类型结构理论。载于编程研讨会：编程会议论文集，法国巴黎，1974年4月9日至11日。408–423。Mads Rosendahl。1989年。自动复杂度分析。载于 FPCA '89：函数式编程语言与计算机体系结构。ACM，144–156。David Sands。1990年。惰性高阶语言的复杂度分析。载于 ESOP '90：第三届欧洲编程研讨会论文集。Springer-Verlag，英国伦敦，361–376。

Lui Sha、Ragunathan Rajkumar 和 John P Lehoczky。1990。优先级继承协议：一种实时同步方法。*IEEE*计算机汇刊39, 9 (1990), 1175–1185。Abraham Silberschatz、Peter Baer Galvin 和 Greg Gagne。2005。操作系统概念（第7版）。Wiley。Kyle Singer、Noah Goldstein、Stefan K. Muller、Kunal Agrawal、I-TING ANGELINA LEE 和 UMUT A. ACAR。2020。面向交互式应用的优先级调度。载于第*32*届 *ACM* 并行算法与架构研讨会论文集。计算机协会，纽约，纽约州，美国，465–477。 https://doi.org/10.1145/3350755.3400236 Frederick Smith、David Walker 和 J. Gregory Morrisett。2000年。别名类型。载于第*9*届欧洲编程语言与系统研讨会论文集（*ESOP '00*）。Springer-Verlag，英国伦敦，英国，366–381。