中国科学技术大学

博士学位论文

并发程序精化验证及其应用

作者姓名： 梁 红 瑾

学科专业： 计算机软件与理论

导师姓名： 冯新宇 教授　邵中 教授

完成时间： 二零一四年五月

University of Science and Technology of China
A dissertation submitted for the degree of PhD

Refinement Verification of Concurrent Programs and Its Applications

Author : Hongjin Liang

Speciality : Computer Software and Theory

Supervisor : Prof. Xinyu Feng & Prof. Zhong Shao

Finished Time : May, 2014

摘 要

许多程序验证问题都可以被归结为精化验证，即证明一个较具体的程序不会比一个较抽象的程序产生更多的行为。本文发掘了在并发环境下精化验证的应用场景，并提出若干种可以支持这些精化应用实例的验证技术。具体来说，本文在理解和验证并发程序精化方面做出了如下贡献。

首先，本文提出了一种基于依赖/保证的模拟关系 RGSim，作为并发程序精化的通用验证手段。这一新颖的模拟关系将线程和并发环境之间的交互作为参数，从而获得可组合性，支持模块化验证。同时，它将精化应用中对并发环境的特定前提参数化，因而具有较好的灵活性和实用性。我们应用 RGSim 验证了并发环境下的几种程序优化。此外，我们还将并发垃圾收集器的验证归结为精化验证，并提出一种基于 RGSim 的通用验证框架。使用这一框架，我们验证了 Boehm 等人提出的并发的标记 -清扫垃圾收集算法。

其次，本文提出了一种霍尔风格的程序逻辑，用于高效地、模块化地验证并发对象的线性一致性。这是并发程序精化验证的一种重要应用。作为该程序逻辑的一部分，我们提出了一种轻量的辅助代码插桩机制。我们的程序逻辑支持可线性化点不固定的并发算法，这些算法的验证难度很大。具体包括：使用帮助机制的无锁算法（如 HSY 栈），可线性化点依赖未来不确定执行的算法（如懒惰集合算法），以及同时有这两种特性的算法（如 RDCSS 算法）。我们还扩展了模拟关系 RGSim 以支持可线性化点不固定的程序，新的模拟关系保证了我们的程序逻辑的可靠性，它可以蕴涵一种上下文精化关系，该精化关系与线性一致性等价。我们应用这一程序逻辑验证了 12 个著名的并发算法，其中两个算法已经在 Java 并发包 `java.util.concurrent` 中使用。

最后，本文展示了一个用精化关系刻画并发对象完整正确性（包括线性一致性以及各种进展性性质）的统一框架。我们证明了对于满足线性一致性的并发对象，每种进展性性质等价于一种特定的对终止性敏感的上下文精化关系。我们用上下文精化关系统一了并发对象的线性一致性和所有常见的进展性性质，包括无等待性、无锁性、无阻碍性、无饥饿性和无死锁性。根据这一结果，对于任何使用并发对象操作的客户端程序，我们可以模块化地验证该客户端程序的安全性和终止性。另一方面，它也告诉我们，有希望借鉴已有的验证上下文精化的技术来同时验证线性一致性和进展性性质。

关键词： 并发 程序验证 精化 模拟关系 程序逻辑 程序正确性

ABSTRACT

Many verification problems can be reduced to refinement verification, i.e., proving that a concrete program has no more behaviors than a more abstract program. This dissertation explores the applications of refinement verification of concurrent programs, and proposes compositional verification techniques that support these applications. It makes several contributions to understanding and verifying concurrent program refinement.

First, it shows a *R*ely-*G*uarantee-based *Sim*ulation (RGSim) as a general proof technique for concurrent program refinement. The novel simulation relation is parameterized with the interference between threads and their parallel environments. It is compositional and supports modular verification. RGSim can incorporate the assumptions about environments made by specific refinement applications, thus is flexible and practical. We apply RGSim to reason about optimizations in parallel contexts. We also reduce the verification of concurrent garbage collectors (GCs) to refinement verification, and propose a general GC verification framework based on RGSim. Using the framework, we verify the Boehm et al. concurrent mark-sweep GC algorithm.

Second, it shows a Hoare-style program logic for modular and effective verification of linearizability of concurrent objects, which is an important application of concurrent program refinement verification. Our logic with a lightweight instrumentation mechanism supports objects with non-fixed linearization points (LPs), including the most challenging ones that use the helping mechanism to achieve lock-freedom (as in HSY elimination-based stack), or have LPs depending on unpredictable future executions (as in the lazy set algorithm), or involve both features (as in the RDCSS algorithm). We generalize RGSim with the support for non-fixed LPs as the meta-theory of our logic, and show it implies a contextual refinement which is equivalent to linearizability. Using our logic we successfully verify 12 well-known algorithms, two of which are in the `java.util.concurrent` package.

Finally, it shows a unified framework that characterizes the full correctness (i.e., linearizability and progress properties) of concurrent objects via contextual refinements. We prove that for linearizable objects, each progress property is equivalent to a certain

form of termination-sensitive contextual refinement. The framework unifies linearizability and all the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. It enables modular verification of safety and liveness properties of client programs, and also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together.

Keywords: Concurrency, Program Verification, Refinement, Simulation, Program Logic, Program Correctness

目 录

索　引 · · ·

插 图

表 格

第 1 章　绪论

程序验证是保证程序的正确性、提高软件的可信度的重要手段。如何准确而直观地刻画程序的正确性，寻找高效而可靠的验证技术，是计算机科学领域被高度关注的古老问题。近年来，随着多核处理器广泛深入到人们的生活中，并发编程逐渐成为软件开发的主流方式，对并发程序验证的研究已越来越重要。

精化验证是程序验证的一个重要分支。程序精化关系（refinement）是建立在两个程序之间的关系。简单来说，如果程序 C 是对程序 $\mathbb{C}$ 的精化，记作 $C \sqsubseteq \mathbb{C}$，那么程序 C 产生的行为是程序 $\mathbb{C}$ 行为的子集。由于 C 不会产生 $\mathbb{C}$ 不具有的行为，因此我们在任何使用程序 $\mathbb{C}$ 的地方，都可以使用程序 C 来替代。当 $\mathbb{C}$ 也是对 C 的精化时，我们可以称 C 和 $\mathbb{C}$ 是等价的。例如，下面这个程序可以看作`x++` 的一种实现：

```
local t;  t := x;  x := t + 1;
```

它先将`x` 的值读入局部变量`t`，再将计算结果写入内存变量`x`。容易看出，若我们不关心局部变量`t` 的值，则这个程序是对`x++` 的精化。反过来，`x++` 也是对这个程序的精化，因此这两个程序是等价的。

研究程序的精化关系及其验证技术不仅仅具有理论上的意义。事实上，许多程序验证问题都可以被归结为精化验证。下面我们列举一些程序精化的典型应用：

- **编译器的正确性刻画和验证**。编译器把源程序 $\mathbb{C}$ 翻译成目标程序 C。一个正确的编译器应该保证 C 不会产生 $\mathbb{C}$ 本身不具有的行为，也就是说 C 应当是 $\mathbb{C}$ 的精化 [1]。若我们能够证明所有目标程序都是对相应的源程序的精化，则我们完成了对编译器正确性的验证。

- **程序或算法的正确性验证**。程序或算法的正确性要求它们满足相应的程序规范（specification）。由于规范往往可以被视为一种更加抽象的程序，我们可以把程序或者算法的正确性刻画为程序和规范之间的精化关系。因此对精化关系的验证实际上是对程序正确性的验证。这种思想的一个关键应用体现在抽象数据类型的实现的正确性验证上（即数据精化）[2]：我们通过证明数据类型操作的具体实现和作为规范的抽象操作之间的精化关系来验证数据类型实现的正确性。

特别地，并发程序的正确性与精化关系有着深刻的联系。例如：

- **并发对象（concurrent object）的实现**：并发对象或并发库向多线程的客户端程序（client program）提供抽象接口，客户端只能通过调用接口方法来访问共享数据结构 [3]。这些接口方法的具体实现往往十分复杂精妙，它们允许细粒度并发操作，以提高多线程访问共享数据结构的效率。客户端程序并不需要知道具体实现，而可以简单地将这些接口方法视作抽象的原子操作（即，其执行不会被其他线程打断）。也就是说，一个正确的并发对象实现应当保证客户端程序能够获得原子地访问数据结构的感受。因此，我们可以将并发对象实现的正确性描述为在并发环境下具体实现与抽象原子操作之间的精化关系。实际上，并发对象的线性一致性（linearizability） [4] ——传统并发对象理论中定义的最常用的功能正确性标准，已被证明与一种上下文精化关系（contextual refinement）等价 [5]。
- **软件事务内存（software transactional memory，简写为 STM）实现算法**： STM 为高层的应用程序员提供事务的概念，程序员可以假设每个事务的原子执行。STM 的实现（如 TL2 算法 [6]）将高层的带事务的代码 $\mathbb{C}$ 转化为底层的、细粒度的不带事务的程序 C。因此，STM 算法的正确性可以归约为这种程序转换的正确性，即保证 C 是 $\mathbb{C}$ 的精化。验证其正确性就是验证这种精化关系。

- **垃圾收集（garbage collection，简写为 GC）算法的正确性刻画和验证**。带有 GC 机制的高级程序语言（如 Java）将应用程序员从手工回收内存空间的繁重工作中解脱出来。程序员编写程序时工作在一个较为抽象的层次，他们感觉不到 GC 的存在。GC 算法将高层的、不做显式内存回收的用户程序（mutator）转化为底层的、可执行的、带 GC 的具体程序。因此，GC 的验证可以归结为这种程序转换的验证，即证明底层具体实现是对高层用户程序的精化。

- **操作系统的正确性刻画和验证**。操作系统本质上为上层应用提供了一个抽象的编程模型，该模型屏蔽了硬件层资源管理的细节，简化了应用程序的开发。那么，操作系统的正确性就要求应用程序在抽象机器模型上的行为和真正的底层硬件机器模型上的行为具有一致性 [7]。操作系统的验证就可以归结为证明两层机器模型之间的精化关系。

由此可见，程序精化关系及其验证技术在程序验证领域有着重要的应用。自上世纪七十年代起，人们开展了大量的程序精化验证工作（如 [2, 8, 9]）。然而

在并发环境下，程序精化关系变得更加复杂，验证更加困难，现有的研究工作尚有许多问题亟待解决。

1.1　关键问题与研究现状概述

本文研究并发程序精化的通用验证技术，并主要关注其在并发对象验证方面的应用。下面仅从这两个方面介绍研究现状与面临的关键问题。

1.1.1　一般并发程序精化的验证

并发程序精化的通用验证技术至少需要满足以下几点要求：

- 不依赖具体的编程语言和机器模型。特别地，应当允许精化关系中的两个程序使用不同的程序语言。这一要求主要考虑对编译器等程序变换的支持。作为编译器输入的源程序往往使用高级程序语言如 C、Java 等，而目标语言一般是汇编等底层程序语言。此时源程序与目标程序所面对的机器模型也不相同，后者往往需要考虑寄存器等底层机器状态，这些在前者所面对的高层编程模型中都被抽象掉了。

- 支持并发粒度不同的程序间的精化。如前所述，程序或算法的正确性可以归结为具体实现 C 与抽象规范 $\mathbb{C}$ 之间的精化。在并发环境下，C 与 $\mathbb{C}$ 往往具有不同的并发粒度。例如，Treiber 栈实现 [10] 采用细粒度并发，入栈操作与出栈操作均分多步完成，线程可以交替执行；而对应的抽象入栈/出栈操作则是原子执行的，具有更粗的并发粒度。为了支持这类程序精化，我们的验证方法不能预设程序的并发粒度，也不能假设所有程序的并发粒度一致。

- 具有可组合性（compositionality）。可组合性是进行模块化验证的重要前提。在并发环境下，我们希望通过验证单个线程上的精化而得到整个多线程并发程序上的精化，即，若我们能证明 $C_1 \sqsubseteq \mathbb{C}_1$ 和 $C_2 \sqsubseteq \mathbb{C}_2$，则可以知道 $C_1 \parallel C_2 \sqsubseteq \mathbb{C}_1 \parallel \mathbb{C}_2$ 成立。这里我们用 $\parallel$ 来表示两个线程的并发组合（parallel composition），$\sqsubseteq$ 是程序精化关系。

然而，已有的程序精化验证技术不能同时达到上述要求。有些工作（如 [11]）建立了程序路径（trace）集合间的包含关系，要求程序使用同样的机器状态。验证上下文等价（contextual equivalence）或上下文精化（contextual refinement）的大量工作假设程序使用相同的编程语言。在组合性的支持方面，传统的精化验证要么是基于完整的封闭系统的，包括串行环境下的工作（如 [1, 2, 8, 12]）；要

么则考虑任意的开放环境（如 [11, 13]）。前者缺少可组合性，无法支持模块化验证。而后者则过于极端，虽然具有很好的可组合性，但它要求程序的精化关系在任何环境下都能保持，这种要求过强，并不适用于上面提到的几类应用，因为这些应用对环境都有或多或少的特定前提假设。我们将在第 2.1节详细解释并发程序精化验证所面临的挑战。

1.1.2　并发对象实现的正确性刻画与验证

如前所述，并发对象实现的验证是并发程序精化验证的一项重要应用。并发对象的功能正确性通常定义为线性一致性（linearizability）[4]。满足线性一致性的并发对象实现应当保证客户端程序在调用对象接口的具体实现时，如同在使用抽象的原子操作。实际上，线性一致性与一种上下文精化关系（contextual refinement）等价 [5]。验证线性一致性就是要证明在并发环境下对象的具体实现是对抽象原子操作的精化。

最常见的验证方法（如见 [14–17]）需要我们找到并发对象的可线性化点（linearization point，简写为 LP）。 LP 是并发对象的操作真正“起作用”（take effect）的时刻。具体来讲，对于并发对象的每个接口方法，LP 是其具体实现代码中的一个特殊的程序步（program step），这一步使得这个方法的效果（如对共享数据结构的修改）可以被并发运行的其他线程所感知。我们要求这样的特殊的程序步必须是唯一的。

然而，在并发对象的实现代码中找到 LP 常常是一件困难的事情。许多无锁（lock-free）并发算法采用帮助（helping）机制（例如 HSY 栈实现算法 [18]），一个方法的 LP 可能在另一个方法的代码中。在这类算法中，每个线程有一个线程描述符，记录自己当前要完成的操作。当线程 1 与线程 2 同时修改某个共享数据而产生访问冲突时，线程 1 可以读取线程 2 的描述符，根据描述符的记录先帮助线程 2 完成它的操作，之后再完成线程 1 自己的操作。在这种情况下，线程 2 的方法在线程 1 的程序步中“起作用”，它的 LP 在线程 1 的代码中，而不在自己的代码中。

除此以外，对于许多乐观（optimistic）算法和懒惰（lazy）算法（例如 Heller 等人提出的懒惰集合算法 [19]），其 LP 的位置可能依赖于未来的某个事件，该事件受不确定的线程间交互所影响，因此无法预知。 这类算法通常允许线程假装不会受到干扰地访问共享数据，之后再确认（validate）刚才的访问是有效的（数据是一致的）。若确认成功，则它结束自己的操作；否则，它回滚至整个操作的开始，重新访问共享数据。在这种情况下，它的 LP 应在之前访问共享数据的时刻，但条件是之后的确认会成功。由于确认成功与否取决于并发环境的影响，

我们不可能静态确定该 LP 的位置。

以上这两类算法被称作 LP 不固定（non-fixed）的算法。对它们的验证是并发对象验证领域中的经典难题。绝大部分已有的工作要么仅能验证 LP 位置固定（即可以在方法的实现代码中静态确定 LP 位置）的算法（如 [14, 16, 20]）；要么虽能支持 LP 不固定的算法，但没有可靠性（soundness）保证（如 [15]）。我们将在第 5.1节详细解释验证 LP 不固定的算法的线性一致性时面临的挑战。

除了应当满足线性一致性这种功能正确性外，并发对象还应当满足进展性（progress）。常见的进展性性质包括无等待性（wait-freedom），无锁性（lock-freedom），无阻碍性（obstruction-freedom），无饥饿性（starvation-freedom）和无死锁性（deadlock-freedom）。前三者针对非阻塞型（non-blocking）并发对象，后两者针对使用锁实现的并发对象。进展性性质定义了对象方法的执行在满足什么样的约束条件下就能终止。例如，无锁性保证无限经常地（infinitely often）有某个方法调用在有限步内终止 [3]。

然而，这些进展性性质很难用于模块化验证。它们的定义（无论是形式化的，还是非形式化的）都是从并发对象的角度出发，描述对象方法执行的终止条件。它们不描述满足某种进展性的并发对象为客户端程序带来的影响。但模块化验证要求我们在验证客户端程序时用抽象原子操作取代并发对象的具体实现，因此我们希望知道在具体实现被抽象操作替代后，客户端程序的行为是否会被影响，以及如何被影响。特别地，针对满足进展性的并发对象，我们关心客户端程序的终止性方面会有怎样的变化。

前面已经提到，已有工作 [5] 证明了并发对象的线性一致性与一种上下文精化关系（contextual refinement）等价。该上下文精化关系描述了客户端程序在输入/输出行为方面受到的影响，但并不考虑终止性方面的影响。Gotsman 和 Yang 发现，若客户端程序在使用抽象操作时终止，则它在使用满足线性一致性和无锁性的具体实现时也一定会终止 [21]。这一工作揭示了无锁性与一种考虑程序终止性的上下文精化关系之间的联系。但还没有工作探讨其他进展性性质对客户端程序的终止性的影响，或将其他进展性性质与精化关系联系起来。

1.2　本文的贡献与组织结构

本文克服了上述各类问题，在理解和验证并发程序精化方面做出了如下贡献。首先，我们提出了一种基于依赖/保证的模拟关系（*R*ely-*G*uarantee-based *Sim*ulation，简称 RGSim），作为并发程序精化的通用验证技术。这一新颖的模拟关系满足第 1.1.1节提到的所有要求。具体表现为以下两个方面：

- RGSim 以依赖/保证条件 [22] 为参数，描述线程和并发环境之间的影响。这使得 RGSim 关系具有并发组合下的可组合性。我们可以应用 RGSim 的可组合性将多线程程序的精化证明分解为单个线程上的精化证明。此外，RGSim 将精化应用中对并发环境的特定前提参数化，因而具有较好的灵活性和实用性。

- RGSim 是一种模拟关系，它关注的是程序的外部可观测行为（如输入/输出事件），允许程序在实现细节上有灵活性。它能够支持低层和高层程序使用不同的编程语言和机器模型，也支持不同粒度的原子操作。

其次，作为 RGSim 的一种重要应用，我们将并发垃圾收集（GC）算法的验证问题归约为程序变换的验证，并由此提出一个通用的 GC 验证框架（定理 4.1），它将传统的一元 Rely-Guarantee 推理 [22] 与基于 RGSim 的关系式证明结合起来。我们用这个框架验证了 Boehm 等人提出的并发垃圾收集算法 [23]。据我们所知，这是第一次形式化证明该算法的正确性。

第三，我们设计了一种霍尔风格的程序逻辑，用以模块化地、高效地验证并发对象的线性一致性。它是第一个具有形式化的可靠性证明、且支持 LP 不固定的并发对象（前面已经提到，这类算法的验证难度很大）的程序逻辑。该逻辑以一元并发程序逻辑 LRG [24] 为基础，但断言和依赖/保证条件都在具体状态和抽象状态的关系上解释。我们还引入了专门验证线性一致性的辅助指令，并为它们设计了新的推理规则。辅助指令将被插桩于对象的具体实现代码中，它们将具体实现与抽象操作联系起来。具体来说，我们通过如下方式支持 LP 不固定的并发对象：

- 对于带有帮助机制的对象，我们引入了 pending 线程池作为辅助状态，它包含的线程的抽象操作可能被其他线程帮助完成。当前线程可以使用辅助指令帮助完成 pending 线程池中的抽象操作。

- 对于依赖未来的 LP，我们引入了 try-commit 机制。**try** 指令投机地执行抽象操作，保留所有可能的情况；之后的 **commit** 指令则在我们知道更多信息时选择正确的投机情况，丢弃错误的投机情况。这种 try-commit 机制让我们无需借助预言变量（prophecy variable）[9, 15] 就可以推理依赖未来的不确定性，而预言变量的现有语义（如 [9]）并不适合霍尔风格的程序验证。

为了证明该程序逻辑的可靠性，我们扩展了模拟关系 RGSim 以支持 LP 不固定的算法。我们的程序逻辑蕴涵这一新的模拟关系，而该模拟关系可以保

证一种上下文精化关系，该上下文精化关系与线性一致性等价。我们应用这一程序逻辑验证了 12 个著名的并发算法，其中两个算法已经在 Java 并发包 `java.util.concurrent` 中使用。

最后，我们提出了一种基于上下文精化的统一框架来刻画并发对象的进展性性质，对于满足线性一致性的并发对象，每种进展性性质等价于一种对终止性敏感的（termination-sensitive）上下文精化关系。该上下文精化框架统一了并发对象的线性一致性和所有常见的进展性性质，包括无等待性、无锁性、无阻碍性、无饥饿性和无死锁性。它一方面揭示出各种进展性性质对客户端程序的终止性的影响，为模块化验证提供了支持。另一方面，它告诉了我们可以用验证上下文精化的方法来同时验证并发对象的线性一致性和进展性。

本文的结构

- 第 2章给出 RGSim 的形式化定义，并证明其具有可组合性。
- 第 3章展示 RGSim 的一些简单的应用，包括推理并发环境下的程序优化，验证抽象操作的细粒度实现等。
- 第 4章提出一种基于 RGSim 的并发 GC 通用验证框架，并验证 Boehm 等人提出的并发的标记 -清扫（mark-sweep）GC 算法 [23]。
- 第 5章介绍验证线性一致性的程序逻辑以及轻量的辅助代码插桩机制，重点讨论对 LP 不固定的并发对象的支持。
- 第 6章展示用于刻画并发对象进展性的上下文精化框架。

第 2 章　验证并发程序精化的模拟关系 RGSim

本章我们提出一种验证并发程序精化的通用技术——模拟关系 RGSim。我们首先讨论并发程序精化验证中的关键挑战，非形式地解释我们的解决方案 RGSim 的基本想法（第 2.1节）。之后，我们给出基本的技术设定，形式化地定义程序精化关系（第 2.2节）。接着，我们给出 RGSim 的定义（第 2.3节），证明其具有可组合性（第 2.4节）。最后我们用一个简单的例子展示 RGSim 的使用方法（第 2.5节）。更多的应用将在第 3章中展示。

2.1　关键挑战与我们的解决方案

并发程序精化验证的主要困难是可组合性。在并发环境下这意味着：若我们证明了单个线程上的精化关系 $C_1 \sqsubseteq \mathbb{C}_1$ 和 $C_2 \sqsubseteq \mathbb{C}_2$，则应该能够知道多线程程序上的精化关系 $C_1 \parallel C_2 \sqsubseteq \mathbb{C}_1 \parallel \mathbb{C}_2$。这里我们用 $\parallel$ 表示并发组合（parallel composition），即两个线程的并发执行。

2.1.1　串行环境下建立的精化关系缺少可组合性

串行程序的行为通常是指它们的控制信息，如终止和异常等，以及它们终止时的程序状态。串行程序间的精化关系 $C \sqsubseteq \mathbb{C}$ 要求程序 C 产生的行为是程序 $\mathbb{C}$ 行为的子集。然而，当引入并发组合以后，这一关系可能无法被保持。

例如，串行环境下正确的编译优化可能改变多线程程序的行为。这是编译器领域里众所周知的一个事实 [25]。展示这一问题的经典例子是 Dekker 的互斥算法，如图 2.1(a) 所示。假设共享变量 `x` 和 `y` 的初值为 0。不难看出该算法确实实现了互斥功能：当两个线程执行到 if 语句的时候，`r1` 和 `r2` 不可能同时为 0，所以最多只有一个线程能进入临界区。由于每个线程的最开始的两条赋值语句之间不具有数据依赖性，很多编译器和处理器都可能会反转它们的执行顺序。这不会改变各个线程自身的行为，但却会导致两个线程的 if 语句判断可能同时为真，从而使得两个线程同时进入临界区，改变整个程序的行为。

除此以外，当 C 与 $\mathbb{C}$ 在原子操作上具有不同的粒度时，二者之间的精化或等价关系在并发组合后也可能无法被保持。在图 2.1(b) 中，下方的两个线程各自与上方的对应线程等价，但下方的整个程序却不是对上方程序的精化。假设 `x` 的初始值为 0，我们知道上方程序执行结束后 `x` 的值只能是 2（这里我们假

```
local r1;              local r2;
x  := 1;               y  := 1;
r1 := y;            ‖  r2 := x;
if (r1 = 0) then       if (r2 = 0) then
  critical region        critical region
```

(a) Dekker 的互斥算法

```
x++; ‖ x++;
```

vs.

```
local r1;          local r2;
r1 := x;        ‖  r2 := x;
x  := r1 + 1;      x  := r2 + 1;
```

(b) 原子操作的不同粒度

图 2.1 引入并发组合后程序等价性丢失

设x++ 是被原子执行的，即它的执行不可被打断），而下方程序则会产生两种可能的结果：1 和 2。这个例子再次说明，串行环境下建立的精化关系在并发环境下缺少可组合性。

2.1.2　假设任意开放环境的精化关系缺乏实用性

串行程序精化关系在并发组合后不能被保持的原因是，它不考虑并发环境对程序行为的影响。并发程序的完全抽象语义（fully abstract semantics）方面的研究工作（如 [11, 13]）给出了另一种定义及验证程序精化的方法。一个程序的语义定义为程序执行路径（trace）的集合。每条执行路径既包含程序自身的状态转换（state transition），也穿插着任意的并发环境的状态转换。程序间的精化关系就定义为相应的执行路径集合间的子集关系。此时我们考虑了环境的行为，因此该精化关系具有很好的可组合性。但是，这种方法要求程序精化关系在任意的环境下都能保持，这种要求太强，对大部分精化应用都不适用。

- 许多提供并发机制的编程语言都要求程序没有数据竞争（data race）。例如，C++ 中，有数据竞争的程序（如图 2.1中的程序）是没有语义的 [26]。因此，相应的编译器也不需要考虑有数据竞争的源程序。我们只需保证当程序的并发环境与程序之间没有数据竞争时，编译过程是正确的。

- 对于并发对象，在证明其具体实现是对抽象原子操作的精化时，我们可以假设并发线程只能通过调用对象的接口来访问共享数据结构。例如，我们只能通过入栈 push 和出栈 pop 这两个操作来修改一个栈对象，栈对象的状态和结构并不是可以任意改变的。

- STM 的实现（如 TL2 [6]）对并发环境也有限制。通常 STM 算法只保证在无数据竞争的情况下事务的实现是正确的。因此，在证明相关的精化关系时，我们可以假设环境与程序之间没有数据竞争。

- 带有 GC 机制的编程语言通常是类型安全的，它们不允许指针算术之类的操作。那么，相应的并发 GC 算法就可以假设并发运行的用户程序满足这些要求。

在以上这些情况中，程序对它的并发环境都有或多或少的特定假设。这些应用并不保证精化关系在任意的环境下都能被保持。

2.1.3　如何允许程序使用不同的编程语言和机器模型？

若我们将精化关系 $C \sqsubseteq \mathbb{C}$ 简单地定义为比较程序 C 和 $\mathbb{C}$ 的状态转换路径，则无法支持 C 和 $\mathbb{C}$ 使用不同的编程语言和机器模型的情况。前面提到，基于完全抽象语义定义的程序精化关系是不实用的，原因之一是它假设了任意开放环境，要求过强；而另一原因是，该精化关系被定义为程序执行路径集合间的包含关系，导致无法支持程序使用不同的编程语言和机器模型。基于同样的理由，验证同一语言中的程序精化或等价关系的大量工作也不具有通用性。

2.1.4　如何定义程序的行为？

在并发环境下，一个程序实际上有两种观测者。除了人类这种外部观测者外，并发的其他线程也是一种观测者。外部观测者们不关心程序的实现细节。对他们来说，程序中的读写操作都是不可观测的，只有输入/输出之类的外部事件才可被观测。然而，对于并发的其他线程，程序中的每一步都可能因为改变共享数据而影响它们的执行，因此对它们来说，这些都是可以观测的。

在定义及验证精化关系时，如果我们仅考虑程序的外部可观测行为，即，精化关系 $C \sqsubseteq \mathbb{C}$ 仅要求程序 C 的外部可观测行为都能由程序 $\mathbb{C}$ 产生，那么该精化关系将失去并发组合下的可组合性，其原因正如我们在第 2.1.1节中所解释的那样。另一方面，如果我们考虑程序的每一步执行，即要求程序 C 的每条执行路径都能由程序 $\mathbb{C}$ 产生，这样得到的精化关系 $C \sqsubseteq \mathbb{C}$ 会太强而丧失实用性。调换程序指令的执行顺序，或是改变程序的原子操作粒度，都可能使这一精化关系不再成立。

2.1.5　我们的解决方案

本文中我们提出了一种基于依赖 -保证的模拟关系（*R*ely-*G*uarantee-based *Sim*ulation，简写为 RGSim）。像通常的弱模拟关系（weak simulation）那样，我们仅保证两个程序的外部可观测行为之间的包含关系，而不要求程序执行路径集合之间具有包含关系。程序中的读写操作都视为不可观测的，从而能够支持程序使用不同的编程语言和机器模型，允许不同的实现细节。

为了获得可组合性，RGSim 显式地描述线程与其并发环境之间的交互。受到并发程序逻辑 Rely-Guarantee [22] 的启发，我们用依赖/保证条件（rely/guarantee conditions）来刻画线程与环境之间的交互。在 Rely-Guarantee 逻辑中，一个线程的依赖条件 R 描述了线程允许环境做怎样的状态转换，保证条件 G 描述了线程自身可能做的状态转换。当两个线程并发执行时，我们需要保证一个线程的保证条件被另一线程的依赖条件所允许。这一要求被称作干涉约束（interference constraint）。若这两个线程满足干涉约束，我们就可以分别验证单个线程，证明在满足依赖条件的环境的影响下线程的行为确实满足其保证条件。这样我们就得到了对线程的并发组合的验证。并发组合后的程序所允许的并发环境应同时是这两个线程所允许的环境，因此其依赖条件是这两个线程的依赖条件的交集；它本身的行为包括两个线程所有可能的行为，因此其保证条件是这两个线程的保证条件的并集。

我们的 RGSim 关系建立在两个程序 C 和 $\mathbb{C}$ 之间，并以它们的依赖/保证条件为参数。其形式为 $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$，其中 $\mathcal{R}$ 和 $\mathcal{G}$ 刻画了 C 与其并发环境之间的交互，$\mathbb{R}$ 和 $\mathbb{G}$ 刻画了 $\mathbb{C}$ 与其并发环境之间的交互。简单来讲，$(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ 的含义是，C 在其环境 $\mathcal{R}$ 影响下的执行不会比 $\mathbb{C}$ 在它的环境 $\mathbb{R}$ 影响下产生更多的外部可观测行为，且 C 和 $\mathbb{C}$ 各自的状态转换分别满足 $\mathcal{G}$ 和 $\mathbb{G}$。这样的 RGSim 关系具有可组合性，在并发组合下的形式为：若 $(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1)$ 和 $(C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2)$ 成立，且干涉约束被满足，即，$\mathcal{G}_2 \subseteq \mathcal{R}_1$，$\mathcal{G}_1 \subseteq \mathcal{R}_2$，$\mathbb{G}_2 \subseteq \mathbb{R}_1$ 和 $\mathbb{G}_1 \subseteq \mathbb{R}_2$ 成立，则我们有

$$(C_1 \parallel C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq (\mathbb{C}_1 \parallel \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2)\,.$$

RGSim 的可组合性给了我们一套验证并发程序精化的证明理论（proof theory）。我们在验证程序精化的时候可以直接应用 RGSim 的可组合性规则，而不需要考虑程序的语义，这对于简化证明过程、促进证明的自动化都有着重要的意义。

此外，RGSim 也与前面提到的假设任意开放环境的精化关系不同。RGSim 允许我们根据具体的应用场景来定制依赖/保证条件 $\mathcal{R}$，$\mathcal{G}$，$\mathbb{R}$ 和 $\mathbb{G}$，将应用中对环境的特定前提参数化，具有较好的灵活性和实用性。例如，若我们想验证

无数据竞争的程序上的编译过程的正确性，我们可以在 $\mathbb{R}$ 和 $\mathbb{G}$ 中描述源程序与其环境之间没有数据竞争。这样，我们就无需考虑图 2.1中的例子，因为图中的两个程序都有数据竞争。

例子　下面我们给一个程序优化的例子来展示 RGSim 的使用。如下所示，源程序 C 经过循环不变代码外提可以得到目标程序 C_1。我们想证明这一优化过程在并发环境下的正确性。完整的形式化证明过程将在第 3.1.2节中展示。

Target Code (C_1)		Source Code (C)
`local t;`		`local t;`
`t := x + 1;`	$\Leftarrow$	`while(i < n) {`
`while(i < n) {`		`  t := x + 1;`
`  i := i + t;`		`  i := i + t;`
`}`		`}`

Benton 证明了在串行环境下优化后的程序 C_1 是对源程序 C 的精化 [27]。在并发环境下，若我们假设任意的环境，则这一优化是不正确的。例如，若并发的其他线程修改 `x`，则程序 C_1 和 C 终止时 `i` 的值可能会不同。事实上，这一优化仅当 C_1 和 C 的并发环境 $\mathcal{R}$ 都不能修改 `x` 和 `t` 的时候才是正确的。C_1 和 C 的保证条件 $\mathcal{G}$ 可以包括任意的状态转换。我们可以证明 RGSim 关系 $(C_1, \mathcal{R}, \mathcal{G}) \preceq (C, \mathcal{R}, \mathcal{G})$ 成立，从而得到这一优化的正确性。

2.2　基本设定

本节我们给出程序语言和机器模型的基本设定。我们的程序语言抽象了大部分语言细节，具有普适性。但为了方便讨论，它包含常见的几种程序结构，如并发组合、顺序组合等。之后我们定义一个通用的、封闭环境下的精化关系 $C \sqsubseteq \mathbb{C}$，它直接要求程序 C 不会比 $\mathbb{C}$ 有更多的可观测行为。简洁而直观的 $\sqsubseteq$ 关系将作为我们的验证目标，我们的 RGSim 关系 $\preceq$（形式化定义在第 2.3节）是 $\sqsubseteq$ 的一种验证手段。

2.2.1　通用的并发程序语言

遵照模拟关系方面的研究工作，我们将程序的语义建模为带标记的转换系统（labeled transition system）。我们首先定义事件和标记。如图 2.2(a) 所示，我们将事件 e 的具体形式抽象掉了，验证程序的人可以根据应用场景来定制它的形式（如输入/输出事件）。这里我们假设事件都是外部可观测的。每个状态转换都有一个标记 ι。标记要么是一个事件，要么是 τ。后者对应于不产生任何事件（即外部不可观测）的状态转换。

$$(\mathit{Events})\quad e \ ::= \ \ldots \qquad\qquad (\mathit{Labels})\ \iota \ ::= \ e \mid \tau$$

(a) 事件及状态转换标记

$$\begin{array}{rrcl}
(\mathit{LState}) & \sigma & ::= & \ldots \\
(\mathit{LExpr}) & E & \in & \mathit{LState} \to \mathit{Int}_\perp \\
(\mathit{LBExp}) & B & \in & \mathit{LState} \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(\mathit{LInstr}) & c & \in & \mathit{LState} \rightharpoonup \mathscr{P}((\mathit{Labels} \times \mathit{LState}) \cup \{\mathbf{abort}\}) \\
(\mathit{LStmt}) & C & ::= & \mathbf{skip} \mid c \mid C_1; C_2 \mid \mathbf{if}\,(B)\,C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\,(B)\,C \mid C_1 \,\|\, C_2 \\
(\mathit{LStep}) & \longrightarrow_L & \in & \mathscr{P}((\mathit{LStmt} \backslash \{\mathbf{skip}\} \times \mathit{LState}) \times \mathit{Labels} \times ((\mathit{LStmt} \times \mathit{LState}) \cup \{\mathbf{abort}\}))
\end{array}$$

(b) 低层语言

$$\begin{array}{rrcl}
(\mathit{HState}) & \Sigma & ::= & \ldots \\
(\mathit{HExpr}) & \mathbb{E} & \in & \mathit{HState} \to \mathit{Int}_\perp \\
(\mathit{HBExp}) & \mathbb{B} & \in & \mathit{HState} \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(\mathit{HInstr}) & \mathbb{c} & \in & \mathit{HState} \rightharpoonup \mathscr{P}((\mathit{Labels} \times \mathit{HState}) \cup \{\mathbf{abort}\}) \\
(\mathit{HStmt}) & \mathbb{C} & ::= & \boldsymbol{skip} \mid \mathbb{c} \mid \mathbb{C}_1;; \mathbb{C}_2 \mid \boldsymbol{if}\ \mathbb{B}\ \boldsymbol{then}\ \mathbb{C}_1\ \boldsymbol{else}\ \mathbb{C}_2 \mid \boldsymbol{while}\ \mathbb{B}\ \boldsymbol{do}\ \mathbb{C} \mid \mathbb{C}_1 \,\|\!|\, \mathbb{C}_2 \\
(\mathit{HStep}) & \longrightarrow_H & \in & \mathscr{P}((\mathit{HStmt} \backslash \{\boldsymbol{skip}\} \times \mathit{HState}) \times \mathit{Labels} \times ((\mathit{HStmt} \times \mathit{HState}) \cup \{\mathbf{abort}\}))
\end{array}$$

(c) 高层语言

图 2.2 通用程序语言

我们定义两个程序语言，分别称作低层语言和高层语言，对应精化关系中的两个程序 C 与 $\mathbb{C}$ 使用的语言。图 2.2(b) 定义了低层语言。我们抽象掉了机器状态、表达式以及原语指令的具体形式。算术表达式 E 被抽象为机器状态到整数的映射，当它在某个状态上不能求值时则映射到 $\perp$。布尔表达式 B 的定义类似。指令 c 被抽象为偏函数，它将初始状态映射到一个转换标记和终止状态的集合，其中转换标记记录了相关状态转换所产生的事件。本文中我们用 $\mathscr{P}(\_)$ 表示幂集。若指令的执行出错（如除零、访问无效地址等），则它将初始状态映射到 **abort**。注意，指令的语义可以是非确定性的（non-deterministic）：从同一初始状态开始可能有不同的执行结果。它也可能在某些初始状态下没有定义，此时该指令被阻塞。例如，当锁 `l` 被其他线程拥有时，获取锁的指令`lock(l)` 就会被阻塞，相应地，`lock(l)` 对这种状态就没有映射。

程序语句是原语指令及它们的组合。**skip** 是一条特殊的语句，可用于指示程序执行的结束，与其他语句顺序组合在一起的时候则表示一个空操作。语句的单步执行表达为带标记的状态转换 $\_ \xrightarrow{\iota}_L \_$，它是一个三元关系，将初始程序配置（即程序语句和机器状态）、标记和终止配置联系起来。当初始语句是 **skip**

$$\frac{(\iota, \Sigma') \in \mathbb{c}\ \Sigma}{(\mathbb{c}, \Sigma) \xrightarrow{\iota} (\textbf{\textit{skip}}, \Sigma')} \qquad \frac{\textbf{abort} \in \mathbb{c}\ \Sigma}{(\mathbb{c}, \Sigma) \longrightarrow \textbf{abort}} \qquad \frac{\Sigma \notin \mathit{dom}(\mathbb{c})}{(\mathbb{c}, \Sigma) \longrightarrow (\mathbb{c}, \Sigma)}$$

$$\frac{}{(\textbf{\textit{skip}} \| \textbf{\textit{skip}}, \Sigma) \longrightarrow (\textbf{\textit{skip}}, \Sigma)} \qquad \frac{(\mathbb{C}_1, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1' \| \mathbb{C}_2, \Sigma')}$$

$$\frac{(\mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_2', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \xrightarrow{\iota} (\mathbb{C}_1 \| \mathbb{C}_2', \Sigma')} \qquad \frac{(\mathbb{C}_1, \Sigma) \longrightarrow \textbf{abort} \quad \text{or} \quad (\mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}$$

图 2.3 通用高层语言的部分操作语义规则

时它没有定义。执行出错时的终止配置为 **abort**。

高层语言的定义与低层语言类似，如图 2.2(c) 所示。注意，高层语言的机器状态和原语指令都可能与低层语言的不同。为了简化之后的讨论，我们让高层语言的复合语句与低层语言的相一致，如 $\mathbb{C}_1;;\mathbb{C}_2$ 和 $\mathbb{C}_1 \| \mathbb{C}_2$ 分别表示 $\mathbb{C}_1$ 与 $\mathbb{C}_2$ 的顺序组合和并发组合。我们的 RGSim 本身并不要求程序语言具有相似的复合结构，但当两层语言的程序结构互相对应时，我们可以更方便地讨论可组合性。

图 2.3给出了高层语言的部分操作语义规则，即 $\_ \xrightarrow{\iota}_H \_$ 的部分定义。本文通常省略 $\_ \xrightarrow{\iota}_H \_$ 和 $\_ \xrightarrow{\iota}_L \_$ 的下标 H 和 L，当箭头上的标记为 τ 时也会省略标记。该高层语言遵循小步操作语义的标准规则，我们仅在图 2.3中展示原语指令和并发组合的规则。需要注意的是，当原语指令 $\mathbb{c}$ 在某个状态 Σ 下被阻塞时（即，$\Sigma \notin \mathit{dom}(\mathbb{c})$），$(\mathbb{c}, \Sigma)$ 走一步将回到它自身。例如，当 $\mathtt{l} \neq 0$ 时（锁 $\mathtt{l}$ 被其他线程所拥有），指令 `lock(l)` 被阻塞，它会重复执行直到 $\mathtt{l}$ 变成 0（锁空闲）；而指令 `unlock(l)` 则简单地将 $\mathtt{l}$ 置为 0，它永远不会被阻塞。本文中所有的原语指令都是被原子地执行的。低层语言的操作语义与高层语言的相似。我们用 $\_ \longrightarrow^* \_$ 表示没有事件产生的零或多步状态转换，用 $\_ \xrightarrow{e}{}^* \_$ 表示产生唯一事件 e 的多步状态转换。

2.2.2　精化关系

下面我们形式化地定义精化关系 $\sqsubseteq$。它比较了低层程序与高层程序产生的事件路径（event trace）。如下所示，事件路径 $\mathcal{E}$ 是由事件构成的序列，其结尾可能是一个特殊的终止记号 **term** 或出错记号 **abort**。本文中我们用 ϵ 表示空序列，用“::”来联结元素与序列。

$$(\mathit{EvtTrace}) \quad \mathcal{E} \ ::= \ \epsilon \mid \textbf{term} \mid \textbf{abort} \mid e :: \mathcal{E}$$

定义 2.1 (事件路径集合)**.** $\mathit{ETrSet}_n(C, \sigma)$ 表示程序 C 从状态 σ 开始执行 n 步所产生的事件路径集合：

1. $ETrSet_0(C,\sigma) \stackrel{\text{def}}{=} \{\epsilon\}$;

2. $ETrSet_{n+1}(C,\sigma) \stackrel{\text{def}}{=}$
$$\begin{aligned}\{\mathcal{E} \mid & (C,\sigma) \longrightarrow (C',\sigma') \wedge \mathcal{E} \in ETrSet_n(C',\sigma') \\ & \vee (C,\sigma) \stackrel{e}{\longrightarrow} (C',\sigma') \wedge \mathcal{E}' \in ETrSet_n(C',\sigma') \wedge \mathcal{E} = e::\mathcal{E}' \\ & \vee (C,\sigma) \longrightarrow \textbf{abort} \wedge \mathcal{E} = \textbf{abort} \\ & \vee C = \textbf{skip} \wedge \mathcal{E} = \textbf{term}\}\,.\end{aligned}$$

我们定义 $ETrSet(C,\sigma)$ 为 $\bigcup_n ETrSet_n(C,\sigma)$.

我们复用上述记号，用 $ETrSet(\mathbb{C},\Sigma)$ 表示高层程序的事件路径集合。注意我们将 **abort** 当作一种特殊的行为，而不将它理解为未定义的任意行为。采取何种理解方式与具体的应用相关。后者这种理解方式更常见于编译器的验证工作中（如 [28]），但本文的基本想法也适用于这种情况，尽管它需要我们对精化关系与模拟关系的定义做出适当改动。

下面我们定义精化关系 $\sqsubseteq$，与 Leroy 定义的精化性质 [1] 相似，它是程序的事件路径集合之间的包含关系。

定义 2.2 (事件路径精化)**.** 我们说 (C,σ) 是对 $(\mathbb{C},\Sigma)$ 的事件路径精化，记作 $(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$，当且仅当

$$ETrSet(C,\sigma) \subseteq ETrSet(\mathbb{C},\Sigma)\,.$$

上述精化关系是定义在两个程序配置之间的，而不单是在程序代码之间。这是因为程序的初始状态也会影响其行为。不妨假设低层程序的初始状态 σ 是由高层程序的初始状态 Σ 经过变换 $\mathbf{T}$ 而得到，即 $\sigma = \mathbf{T}(\Sigma)$。那么，我们可以定义 $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$：

$$C \sqsubseteq_{\mathbf{T}} \mathbb{C} \;\stackrel{\text{def}}{=}\; \forall \sigma, \Sigma.\, \sigma = \mathbf{T}(\Sigma) \implies (C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)\,.$$

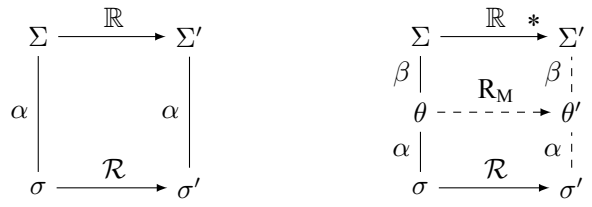
若 $\mathbf{T}$ 同时也是程序之间的变换，即 $C = \mathbf{T}(\mathbb{C})$，则我们可以用精化来定义 $\mathbf{T}$ 的正确性：

$$\mathsf{Correct}(\mathbf{T}) \;\stackrel{\text{def}}{=}\; \forall C, \mathbb{C}.\, C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq_{\mathbf{T}} \mathbb{C}\,. \tag{2.1}$$

在事件路径精化的基础上我们可以定义事件路径等价关系，它要求两个方向的精化同时成立：

$$(C,\sigma) \approx (\mathbb{C},\Sigma) \;\stackrel{\text{def}}{=}\; (C,\sigma) \sqsubseteq (\mathbb{C},\Sigma) \wedge (\mathbb{C},\Sigma) \sqsubseteq (C,\sigma)\,.$$

然后我们可以定义 $C \approx_{\mathbf{T}} \mathbb{C}$ 为 $\forall \sigma, \Sigma.\, \sigma = \mathbf{T}(\Sigma) \Longrightarrow (C,\sigma) \approx (\mathbb{C},\Sigma)$。



(a) α-相关的状态转换　　(b) TRANS 规则的附加条件

图 2.4 高低层相关联的状态转换

2.3　RGSim 的定义

事件路径精化关系直接定义在程序的外部可观测行为上。它直观、简单，且不依赖于具体的编程语言细节。然而，正如我们在第 2.1节中所解释的，事件路径精化关系在并发组合下不具有可组合性。本节我们提出具有可组合性的 RGSim 关系，作为事件路径精化的证明手段。

RGSim 关系的形式为 $(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$。它是建立在程序配置 (C,σ) 和 $(\mathbb{C},\Sigma)$ 之间的、余归纳（co-inductive）定义的弱模拟关系（weak simulation）。同时，它以两个程序的依赖/保证条件为参数。依赖/保证条件是状态上的二元关系：

$$\mathcal{R},\mathcal{G} \in \mathscr{P}(LState \times LState), \quad \mathbb{R},\mathbb{G} \in \mathscr{P}(HState \times HState).$$

RGSim 关系还带有两个额外的参数：不变条件 α 和后条件 γ，二者都是低层状态与高层状态之间的关系：

$$\alpha,\gamma \in \mathscr{P}(LState \times HState).$$

RGSim 关系的形式化定义将在定义 2.4中给出。在此之前我们需要先定义 α-相关的状态转换。

定义 2.3 (α-相关的状态转换)**.** $\langle\mathcal{R},\mathbb{R}\rangle_\alpha \stackrel{\text{def}}{=}$

$$\{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid (\sigma,\sigma') \in \mathcal{R} \wedge (\Sigma,\Sigma') \in \mathbb{R} \wedge (\sigma,\Sigma) \in \alpha \wedge (\sigma',\Sigma') \in \alpha\}.$$

如图 2.4(a) 所示，$\mathcal{R}$ 和 $\mathbb{R}$ 的 α-相关的状态转换集合 $\langle\mathcal{R},\mathbb{R}\rangle_\alpha$ 包括了 $\mathcal{R}$ 和 $\mathbb{R}$ 中的所有被 α 联系起来的对应状态转换。$\langle\mathcal{G},\mathbb{G}\rangle_\alpha$ 的定义与此类似。

定义 2.4 (RGSim)**.** *RGSim* 是满足下述性质的最大关系。若 $(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$，则 $(\sigma,\Sigma) \in \alpha$ 且下述全部成立：

1. 若 $(C,\sigma) \longrightarrow (C',\sigma')$，则存在 $\mathbb{C}'$ 和 Σ' 使得 $(\mathbb{C},\Sigma) \longrightarrow^* (\mathbb{C}',\Sigma')$，$((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle\mathcal{G},\mathbb{G}^*\rangle_\alpha$ 和 $(C',\sigma',\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}',\Sigma',\mathbb{R},\mathbb{G})$ 成立；

$$
\begin{tikzcd}
(\mathbb{C},\Sigma) \arrow[r, dashed, "e", "\mathbb{G}"', "*"] \arrow[d, dash, "\preceq"', "\alpha"] & (\mathbb{C}',\Sigma') \arrow[d, dashed, "\preceq"', "\alpha"] \\
(C,\sigma) \arrow[r, "e", "\mathcal{G}"'] & (C',\sigma')
\end{tikzcd}
\qquad
\begin{tikzcd}
(\mathbb{C},\Sigma) \arrow[r, "*", "\mathbb{R}"'] \arrow[d, dash, "\preceq"', "\alpha"] & (\mathbb{C}',\Sigma') \arrow[d, dashed, "\preceq"', "\alpha"] \\
(C,\sigma) \arrow[r, "\mathcal{R}"'] & (C',\sigma')
\end{tikzcd}
$$

(a) 程序自身　　(b) 环境

图 2.5 RGSim 的模拟关系图

2. 若 $(C,\sigma) \xrightarrow{e} (C',\sigma')$，则存在 $\mathbb{C}'$ 和 Σ' 使得 $(\mathbb{C},\Sigma) \xrightarrow{e}{}^{*} (\mathbb{C}',\Sigma')$，$((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle \mathcal{G},\mathbb{G}^{*}\rangle_{\alpha}$ 和 $(C',\sigma',\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}',\Sigma',\mathbb{R},\mathbb{G})$ 成立；

3. 若 $C = \textbf{skip}$，则存在 Σ' 使得 $(\mathbb{C},\Sigma) \longrightarrow^{*} (\textbf{\textit{skip}},\Sigma')$，$((\sigma,\sigma),(\Sigma,\Sigma')) \in \langle \mathcal{G},\mathbb{G}^{*}\rangle_{\alpha}$，$(\sigma,\Sigma') \in \gamma$ 和 $\gamma \subseteq \alpha$ 成立；

4. 若 $(C,\sigma) \longrightarrow \textbf{abort}$，则 $(\mathbb{C},\Sigma) \longrightarrow^{*} \textbf{abort}$ 成立；

5. 若 $((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle \mathcal{R},\mathbb{R}^{*}\rangle_{\alpha}$，则 $(C,\sigma',\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma',\mathbb{R},\mathbb{G})$ 成立。

在此基础上，我们定义 $(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C},\mathbb{R},\mathbb{G})$ 当且仅当对于任意的 σ 和 Σ，若 $(\sigma,\Sigma) \in \zeta$，则有 $(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$。这里前条件 $\zeta \in \mathscr{P}(LState \times HState)$ 用来联系初始状态 σ 和 Σ.

简单来讲，$(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$ 表示，无论环境 $\mathcal{R}$ 和 $\mathbb{R}$ 如何干扰，低层程序配置 (C,σ) 的执行是对高层程序配置 $(\mathbb{C},\Sigma)$ 执行的模拟，且二者的行为分别保证 $\mathcal{G}$ 和 $\mathbb{G}$。具体地说，它要求 C 的所有执行都满足下列条件：

- 从 α-相关的状态出发，C 的每一步执行都对应于 $\mathbb{C}$ 的零或多步执行，且执行之后的状态也是 α-相关的。如果 C 的这一步执行产生了一个外部可观测事件，则对应的 $\mathbb{C}$ 的若干步执行也产生同样的事件。图 2.5(a) 展示了程序步产生事件时的模拟关系图。图中我们遵照 Leroy 的标记方法 [1]，用实线表示前提，用虚线表示结论。

- 关系 α 反映了低层机器状态对高层状态的抽象。我们要求它在两层程序对应的执行过程中被保持，因此将它称作“不变条件”。例如，集合的实现算法可能使用一个链表作为集合在机器内存中的具体表示，那么，α 就可以定义为该链表和一个抽象的数学集合之间的关系，前者是低层机器状态，后者是对应的高层状态。

- C 和 $\mathbb{C}$ 的对应的状态转换应当满足 $\langle \mathcal{G},\mathbb{G}^{*}\rangle_{\alpha}$。也就是说，$C$ 的每一步状态转换应满足其保证条件 $\mathcal{G}$，对应的 $\mathbb{C}$ 的若干步状态转换应满足 $\mathbb{G}$ 的传

$$\mathsf{InitRel}_{\mathbf{T}}(\zeta) \stackrel{\text{def}}{=} \forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies (\sigma, \Sigma) \in \zeta$$

$$B \Leftrightarrow \mathbb{B} \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid B\ \sigma = \mathbb{B}\ \Sigma\} \qquad B \Lleftarrow \mathbb{B} \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid B\ \sigma \wedge \mathbb{B}\ \Sigma\}$$

$$\mathsf{Intuit}(\alpha) \stackrel{\text{def}}{=} \forall \sigma, \Sigma, \sigma', \Sigma'.\ (\sigma, \Sigma) \in \alpha \wedge \sigma \subseteq \sigma' \wedge \Sigma \subseteq \Sigma' \implies (\sigma', \Sigma') \in \alpha$$

$$\alpha \uplus \beta \stackrel{\text{def}}{=} \{(\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2) \mid (\sigma_1, \Sigma_1) \in \alpha \wedge (\sigma_2, \Sigma_2) \in \beta\} \qquad \eta \# \alpha \stackrel{\text{def}}{=} (\eta \cap \alpha) \subseteq (\eta \uplus \alpha)$$

$$\beta \circ \alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \exists \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\} \qquad \alpha^{-1} \stackrel{\text{def}}{=} \{(\Sigma, \sigma) \mid (\sigma, \Sigma) \in \alpha\}$$

$$\mathsf{Id} \stackrel{\text{def}}{=} \{(\sigma, \sigma) \mid \sigma \in \mathit{LState}\} \qquad \mathsf{True} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma, \sigma' \in \mathit{LState}\}$$

$$\begin{aligned}
&\mathrm{R_M}\ \mathsf{isMidOf}\ (\alpha, \beta; \mathcal{R}, \mathbb{R}) \stackrel{\text{def}}{=} \\
&\quad \forall \sigma, \sigma', \Sigma, \Sigma'.\ ((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R} \rangle_{\beta \circ \alpha} \implies \forall \theta.\ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta \implies \\
&\quad \exists \theta'.\ ((\sigma, \sigma'), (\theta, \theta')) \in \langle \mathcal{R}, \mathrm{R_M} \rangle_{\alpha} \wedge ((\theta, \theta'), (\Sigma, \Sigma')) \in \langle \mathrm{R_M}, \mathbb{R} \rangle_{\beta}
\end{aligned}$$

图 2.6 RGSim 有关的辅助定义

递闭包。保证条件 $\mathcal{G}$ 和 $\mathbb{G}$ 是对程序自身行为的一种抽象，之后我们讨论图 2.7中的 PAR 规则时会看到，当引入并发组合后，$\mathcal{G}$ 和 $\mathbb{G}$ 也将充当环境线程的依赖条件。注意这里我们不要求 $\mathbb{C}$ 的每一步都必须满足它的 $\mathbb{G}$（我们可以这样做，但并不需要强制这样做），我们会在第 2.4节用一个例子 (2.2) 式详细地解释这件事。

- 如果 C 终止，那么 $\mathbb{C}$ 也终止，且它们的终止状态满足后条件 γ。我们要求 $\gamma \subseteq \alpha$，即，终止状态之间的关系也满足不变条件。

- 只有当 $\mathbb{C}$ 的执行出错（不安全）时，C 才可以出错。也就是说，安全的高层程序对应的低层程序也必须是安全的。

- 无论低层环境 $\mathcal{R}$ 与高层环境 $\mathbb{R}$ 做什么，只要其状态转换是 α-相关的，就不能影响 C 与 $\mathbb{C}$ 之间的模拟关系。图 2.5(b) 展示了与环境有关的模拟关系图。这里，$\mathcal{R}$ 的一步可能对应 $\mathbb{R}$ 的零或多步。但是，与程序步不同，$\mathcal{R}$ 的某些步可能并不对应 $\mathbb{R}$ 的状态转换。另一方面，仅仅要求 $\mathcal{R}$ 的每一步都对应 $\mathbb{R}$ 的若干步（或者说，$\mathcal{R}$ 是对 $\mathbb{R}$ 的模拟，见第 2.4节的 (2.3) 式）并不足以保证整个模拟关系的并发可组合性，具体原因我们将在第 2.4节解释。

在上述模拟关系的基础上，我们用前条件 ζ 隐藏初始状态，从而定义程序代码之间的 RGSim 关系 $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$。由定义可知，$\zeta \subseteq \alpha$，即，程序的初始状态应当也满足不变条件。在实际应用中 α 通常是很弱的，可以自然地被 ζ 和 γ 所蕴涵，事实上在第 3章的大部分例子中，α，ζ 和 γ 三者是完全相同的。

RGSim 关系对于事件路径精化（定义 2.2）是可靠的，即，$(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$ 保证了 (C,σ) 不会比 $(\mathbb{C},\Sigma)$ 产生更多的外部可观测行为。

定理 2.5 (RGSim 的可靠性/适当性)**.** *若存在 $\mathcal{R}$,$\mathcal{G}$,$\mathbb{R}$,$\mathbb{G}$,α 和 γ 使得 $(C,\sigma,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C},\Sigma,\mathbb{R},\mathbb{G})$ 成立，则有 $(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$。*

上述可靠性（soundness）或适当性（adequacy）定理说明，RGSim 是精化关系 $\sqsubseteq$ 的一种证明手段，后者简单而直观，是我们最终所关心的性质。为了证明该定理，我们可以先加强依赖条件的限制，使之仅包含同一（identity）的状态转换；放宽保证条件，使之包含任意的状态转换。然后我们证明这样的模拟关系蕴涵事件路径精化关系。该定理的证明过程已经在证明辅助工具 Coq [29] 中实现 [30]。

当低层程序的初始状态是由高层程序的初始状态经过变换 **T** 而得到的时，我们可以有下面的推论 2.6成立。其中，$\mathsf{InitRel}_{\mathbf{T}}(\zeta)$（定义在图 2.6中）表示状态上的变换 **T** 能够保证前条件 ζ.

推论 2.6. *若存在 $\mathcal{R}$,$\mathcal{G}$,$\mathbb{R}$,$\mathbb{G}$,α,ζ 和 γ 使得 $\mathsf{InitRel}_{\mathbf{T}}(\zeta)$ 和 $(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\times\gamma} (\mathbb{C},\mathbb{R},\mathbb{G})$ 成立，则有 $C \sqsubseteq_{\mathbf{T}} \mathbb{C}$。*

2.4　RGSim 的可组合性

RGSim 关系在各种程序结构包括并发组合上都具有可组合性。图 2.7展示了 RGSim 的可组合性规则，它们构成了一套验证并发程序精化的关系式证明理论。

正如 Rely-Guarantee 逻辑 [22] 那样，我们要求前后条件在并发环境的干扰下稳定（stable）。这里我们考虑的是关系 ζ 在状态转换对的集合 Λ 下的稳定性，其中 $\Lambda \in \mathscr{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))$。

定义 2.7 (稳定性)**.** $\mathsf{Sta}(\zeta,\Lambda)$ *成立，当且仅当*
对于任意 σ，σ'，Σ 和 Σ'，若 $(\sigma,\Sigma) \in \zeta$ 且 $((\sigma,\sigma'),(\Sigma,\Sigma')) \in \Lambda$，那么 $(\sigma',\Sigma') \in \zeta$。

通常我们需要 $\mathsf{Sta}(\zeta,\langle\mathcal{R},\mathbb{R}^*\rangle_\alpha)$，即，若初始状态满足 ζ，那么当 $\mathcal{R}$ 和 $\mathbb{R}^*$ 做了相关的状态转换后，结束状态仍然要满足 ζ。展开 $\langle\mathcal{R},\mathbb{R}^*\rangle_\alpha$ 的定义后，我们可以知道 α 自身在任何 α-相关的状态转换下稳定，即 $\mathsf{Sta}(\alpha,\langle\mathcal{R},\mathbb{R}^*\rangle_\alpha)$ 恒成立。下面这个例子中，我们可以证明 $\mathsf{Sta}(\zeta,\langle\mathcal{R},\mathbb{R}^*\rangle_\alpha)$ 成立。此时环境增加 x 的值，我们将一元情况下稳定的断言 $\mathrm{x} \geq 0$ 提升（lift）为二元关系 ζ：

$$\zeta \stackrel{\text{def}}{=} \{(\sigma,\Sigma) \mid \sigma(\mathrm{x}) = \Sigma(\mathrm{x}) \wedge \sigma(\mathrm{x}) \geq 0\} \qquad \alpha \stackrel{\text{def}}{=} \{(\sigma,\Sigma) \mid \sigma(\mathrm{x}) = \Sigma(\mathrm{x})\}$$
$$\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma' = \sigma\{\mathrm{x} \rightsquigarrow \sigma(\mathrm{x})+1\}\} \qquad \mathbb{R} \stackrel{\text{def}}{=} \{(\Sigma,\Sigma') \mid \Sigma' = \Sigma\{\mathrm{x} \rightsquigarrow \Sigma(\mathrm{x})+1\}\}$$

$$\frac{\zeta \subseteq \alpha}{(\textbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\zeta} (\textbf{\textit{skip}}, \mathbb{R}, \mathsf{Id})} \ (\mathsf{SKIP})$$

$$\frac{(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \qquad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma\ltimes\eta} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})}{(C_1; C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\eta} (\mathbb{C}_1;;\mathbb{C}_2, \mathbb{R}, \mathbb{G})} \ (\mathsf{SEQ})$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1\ltimes\gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \qquad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2\ltimes\gamma} (\mathbb{C}_2, \mathbb{R}, \mathbb{G}) \\ \zeta \subseteq (B \Leftrightarrow \mathbb{B}) \qquad \zeta_1 = (\zeta \cap (B \wedge\!\!\!\wedge \mathbb{B})) \qquad \zeta_2 = (\zeta \cap (\neg B \wedge\!\!\!\wedge \neg\mathbb{B})) \qquad \zeta \subseteq \alpha\end{array}}{(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{\textit{if}}\ \mathbb{B}\ \textbf{\textit{then}}\ \mathbb{C}_1\ \textbf{\textit{else}}\ \mathbb{C}_2, \mathbb{R}, \mathbb{G})} \ (\mathsf{IF})$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma_1\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \gamma \subseteq (B \Leftrightarrow \mathbb{B}) \qquad \gamma_1 = (\gamma \cap (B \wedge\!\!\!\wedge \mathbb{B})) \qquad \gamma_2 = (\gamma \cap (\neg B \wedge\!\!\!\wedge \neg\mathbb{B}))\end{array}}{(\textbf{while}\ (B)\ C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma\ltimes\gamma_2} (\textbf{\textit{while}}\ \mathbb{B}\ \textbf{\textit{do}}\ \mathbb{C}, \mathbb{R}, \mathbb{G})} \ (\mathsf{WHILE})$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma_1} (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1) \qquad (C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\zeta\ltimes\gamma_2} (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2) \\ \mathcal{G}_1 \subseteq \mathcal{R}_2 \qquad \mathcal{G}_2 \subseteq \mathcal{R}_1 \qquad \mathbb{G}_1 \subseteq \mathbb{R}_2 \qquad \mathbb{G}_2 \subseteq \mathbb{R}_1\end{array}}{(C_1 \parallel C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq_{\alpha;\zeta\ltimes(\gamma_1\cap\gamma_2)} (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2)} \ (\mathsf{PAR})$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad (\zeta \cup \gamma) \subseteq \alpha' \subseteq \alpha \qquad \mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha)}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \ (\mathsf{STREN}\text{-}\alpha)$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \alpha \subseteq \alpha' \qquad \mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R}^* \rangle_{\alpha'})}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \ (\mathsf{WEAKEN}\text{-}\alpha)$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \zeta' \subseteq \zeta \qquad \gamma \subseteq \gamma' \subseteq \alpha \qquad \mathcal{R}' \subseteq \mathcal{R} \qquad \mathbb{R}' \subseteq \mathbb{R} \qquad \mathcal{G} \subseteq \mathcal{G}' \qquad \mathbb{G} \subseteq \mathbb{G}'\end{array}}{(C, \mathcal{R}', \mathcal{G}') \preceq_{\alpha;\zeta'\ltimes\gamma'} (\mathbb{C}, \mathbb{R}', \mathbb{G}')} \ (\mathsf{CONSEQ})$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \eta \subseteq \beta \qquad \eta \# \{\zeta, \gamma, \alpha\} \\ \mathsf{Intuit}(\{\alpha, \zeta, \gamma, \beta, \eta, \mathcal{R}, \mathbb{R}, \mathcal{R}_1, \mathbb{R}_1\}) \qquad \mathsf{Sta}(\eta, \{\langle \mathcal{G}, \mathbb{G}^* \rangle_\alpha, \langle \mathcal{R}_1, \mathbb{R}_1^* \rangle_\beta\})\end{array}}{(C, \mathcal{R} \uplus \mathcal{R}_1, \mathcal{G} \uplus \mathcal{G}_1) \preceq_{\alpha\uplus\beta;(\zeta\uplus\eta)\ltimes(\gamma\uplus\eta)} (\mathbb{C}, \mathbb{R} \uplus \mathbb{R}_1, \mathbb{G} \uplus \mathbb{G}_1)} \ (\mathsf{FRAME})$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathrm{C_M}, \mathrm{R_M}, \mathrm{G_M}) \\ (\mathrm{C_M}, \mathrm{R_M}, \mathrm{G_M}) \preceq_{\beta;\delta\ltimes\eta} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \qquad \mathrm{R_M}\ \mathsf{isMidOf}\ (\alpha, \beta; \mathcal{R}, \mathbb{R}^*)\end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\beta\circ\alpha;(\delta\circ\zeta)\ltimes(\eta\circ\gamma)} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \ (\mathsf{TRANS})$$

图 2.7 RGSim 的可组合性规则

对于图 2.7中的每条规则，我们假设前后条件在有关环境下都是稳定的（作为隐式附加条件）。例如，在 SKIP 规则中，我们假设 $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R}^* \rangle_\alpha)$ 成立。此外，我们还隐式假设所有的依赖/保证条件都包含同一（identity）的状态转换，这样假设不会对程序的外部可观测行为造成影响。

图 2.7中，SKIP，SEQ，IF 和 WHILE 规则与霍尔逻辑（Hoare logic）中的对应推理规则十分相似。SEQ 规则中，γ 既是 C_1 和 $\mathbb{C}_1$ 的后条件，也是 C_2 和 $\mathbb{C}_2$ 的前条件。IF 规则要求在满足前条件 ζ 的状态下，低层与高层的 if 语句的布

尔条件求值相等。这里用到的集合 $B \Leftrightarrow \mathbb{B}$ 和 $B \wedge\!\!\!\wedge \mathbb{B}$ 在图 2.6中定义。此外，前条件 ζ 应当蕴涵不变条件 α。WHILE 规则中，关系 γ 是循环不变式，在每次循环迭代的开始都应当成立。与 IF 规则类似，WHILE 规则要求 γ 保证 $B \Leftrightarrow \mathbb{B}$。

并发可组合性　图 2.7的 PAR 规则展示了 RGSim 在并发组合下的可组合性。这条规则要求干涉约束（interference constraint）成立，即，只有当每个线程的保证条件蕴涵另一个线程的依赖条件时，两个线程才可以并发组合起来。并发组合后的程序所依赖的环境是两个线程的公共环境，所保证的行为包含两个线程各自的行为。

注意，虽然 RGSim 并不要求高层程序的每一步都满足其保证条件（见定义 2.4的前两条），但这并不影响 RGSim 的并发可组合性。这是因为精化关系（以及验证精化的模拟关系）允许低层程序比高层程序的行为更少。当证明 $C_1 \parallel C_2$ 是对 $\mathbb{C}_1 \lll \mathbb{C}_2$ 的精化（或模拟）时，我们只需找到 $\mathbb{C}_1$ 和 $\mathbb{C}_2$ 交替执行过程的一个子集，使得 C_1 和 C_2 的交替执行是对它的精化（或模拟）。那么，高层的依赖/保证条件也只需保证这个交替执行的子集的存在性。下面我们通过一个简单的例子来解释这件事。我们可以证明下式成立：

$$(\texttt{x:=x+2}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\texttt{x:=x+1;x:=x+1}, \mathbb{R}, \mathbb{G}), \tag{2.2}$$

其中 $\mathcal{R}$，$\mathcal{G}$，$\mathbb{R}$ 和 $\mathbb{G}$ 允许 x 增加 2；α，ζ 和 γ 将低层和高层的 x 联系起来：

$$\begin{aligned}
\mathcal{R} = \mathcal{G} &\stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' = \sigma \vee \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 2\}\}\,; \\
\mathbb{R} = \mathbb{G} &\stackrel{\text{def}}{=} \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma \vee \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 2\}\}\,; \\
\alpha = \zeta = \gamma &\stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}\,.
\end{aligned}$$

注意到高层程序实际上比它的 $\mathbb{G}$ 具有更细的粒度，但是在证明 (2.2) 式的模拟关系时，我们只需要用到高层程序的完全不被环境影响就执行到结束的这种执行过程。我们还可以证明 $(\texttt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\texttt{print(x)}, \mathbb{R}, \mathbb{G})$。这里我们用 **print**$(E)$ 指令来观测 x 的值，该指令会产生一个外部可观测事件 **out**(n)（其中 E 求值得到 n）。那么，应用 PAR 规则后，我们就得到：

$$(\texttt{x:=x+2} \parallel \texttt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\texttt{(x:=x+1;x:=x+1)} \lll \texttt{print(x)}, \mathbb{R}, \mathbb{G})\,.$$

上式并不违反我们对程序精化的直观认识。低层程序产生的所有可观测事件都可在高层产生，尽管后者具有更细的粒度而有更多的可观测行为。

RGSim 定义的另一微妙之处在定义 2.4的第 5 条，与环境有关的条件。这一条件是 RGSim 具有并发可组合性的关键之一。可能有读者会认为这一条件不够

自然，更自然的条件应当要求 $\mathcal{R}$ 是对 $\mathbb{R}$ 的模拟，即用下面的条件取代定义 2.4的第 5 条：

$$\begin{aligned}&\text{若}(\sigma,\sigma')\in\mathcal{R}\text{，则存在}\Sigma'\text{使得}(\Sigma,\Sigma')\in\mathbb{R}^*\\&\text{和}(C,\sigma',\mathcal{R},\mathcal{G})\preceq'_{\alpha;\gamma}(\mathbb{C},\Sigma',\mathbb{R},\mathbb{G})\text{成立。}\end{aligned}\tag{2.3}$$

我们用 $\preceq'$ 来指代改动后的模拟关系。然而，$\preceq'$ 并不具有并发可组合性。下面我们举一个反例。设不变条件 α 要求低层的 x 的值不比高层的 x 大，即

$$\alpha \stackrel{\text{def}}{=} \{(\sigma,\Sigma)\mid\sigma(\mathtt{x})\le\Sigma(\mathtt{x})\},$$

那么，我们可以证明：

$$(\mathtt{x:=x+1},\ \mathsf{Id},\ \mathsf{True})\preceq'_{\alpha;\alpha\ltimes\alpha}(\mathtt{x:=x+2},\ \mathsf{Id},\ \mathsf{True});\tag{2.4}$$

$$(\mathtt{x:=0;print(x)},\ \mathsf{True},\ \mathsf{Id})\preceq'_{\alpha;\alpha\ltimes\alpha}(\mathtt{x:=0;print(x)},\ \mathsf{True},\mathsf{Id}).\tag{2.5}$$

这里我们用 Id 和 True 分别表示同一的状态转换的集合与任意转换的集合。二者在图 2.6中定义，并且我们把低层定义的记号复用在高层。但是，并发组合后的模拟关系（即下式）不成立：

$$\begin{aligned}&(\mathtt{x:=x+1}\parallel(\mathtt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True})\\&\preceq'_{\alpha;\alpha\ltimes\alpha}(\mathtt{x:=x+2}\parallel\!\mid(\mathtt{x:=0;print(x)}),\ \mathsf{Id},\ \mathsf{True}).\end{aligned}$$

这是因为 $\mathcal{R}$（或 $\mathbb{R}$）仅仅是对某个线程 t 所允许的所有环境行为的一种抽象。对于任何线程 t′，只要其行为被 $\mathcal{R}$（或 $\mathbb{R}$）所允许，就可以与线程 t 并发执行。因此，为了得到并发可组合性，我们必须确保在任何可能的并发线程 t′ 的影响下模拟关系都能被保持。若我们用原来的定义 $\preceq$，则 (2.5) 式不成立：在环境的 α-相关的状态转换的干扰下，低层程序打印出的数可能会比高层打印的更小。

其他规则　我们还为 RGSim 设计了其他一些有用的规则。例如，STREN-α 规则允许我们把不变关系 α 替换为一个更强的不变关系 α'。我们需要检查 α' 确实是在 α-相关的程序步下被保持的不变关系，即 $\mathsf{Sta}(\alpha',\langle\mathcal{G},\mathbb{G}^*\rangle_\alpha)$ 成立。与之对称的 WEAKEN-α 规则要求 α 在 α'-相关的环境步下被保持，这样我们就可以将 α 替换为一个更弱的不变关系 α'。此外，像 Rely-Guarantee 逻辑 [22] 一样，我们可以用 CONSEQ 规则来强化或弱化前后条件和依赖/保证条件。

FRAME 规则让我们可以做局部推理（local reasoning）[31]。当证明 C 和 $\mathbb{C}$ 之间的模拟关系时，α，ζ 和 γ 只需描述 C 和 $\mathbb{C}$ 用到的局部资源，$\mathcal{R}$，$\mathcal{G}$，$\mathbb{R}$ 和 $\mathbb{G}$ 也只需描述局部资源上的状态转换。系统中可能还含有额外的资源 η，可被其他程序访问。设对 η 的访问保持 β 且遵守 $\mathcal{R}_1$，$\mathcal{G}_1$，$\mathbb{R}_1$ 和 $\mathbb{G}_1$。那么，我们就可以将使用局部资源 α 的模拟关系的证明复用在含有额外资源 η 的上下文中。图 2.6给

出了这条规则用到的辅助定义。我们将一元情况下在状态之间定义的不相交并（disjoint union）操作 $\uplus$ 提升（lift）到状态对（pair）上定义。Intuit(α) 表示 α 是直觉性的（intuitionistic），在扩展状态时它可以被保持。不相交性 $\eta \# \alpha$ 要求任何同时满足 η 和 α 的状态对都可以被分成两个不相交的状态对，它们分别满足 η 和 α。举例来说，设 $\eta \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{y}) = \Sigma(\mathtt{y})\}$ 和 $\alpha \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$（其中 x and y 是两个不同的变量），则 η 和 α 都是直觉性的，且 $\eta \# \alpha$ 成立。此外，FRAME 规则还要求额外资源 η 在程序自身以及额外环境的执行下都是稳定的。我们将 $(\eta \# \zeta) \wedge (\eta \# \gamma) \wedge (\eta \# \alpha)$ 简记为 $\eta \# \{\zeta, \gamma, \alpha\}$。该规则中还使用了其他类似的记法。

最后，TRANS 规则表达了 RGSim 的传递性，它允许我们引入一个中间层作为桥梁来验证低层和高层程序之间的精化。我们应精心选择中间层环境 $\mathrm{R_M}$。如图 2.4(b) 所示，好的 $\mathrm{R_M}$ 让我们能够将 $(\beta \circ \alpha)$-相关的状态转换分解成 β-相关的和 α-相关的状态转换。这里 $\circ$ 定义了两个二元关系的合成关系，isMidOf 是上述对 $\mathrm{R_M}$ 的约束条件，二者均定义在图 2.6中。我们用 θ 来表示中间层的状态。

可靠性　图 2.7的所有规则都是可靠的。这里可靠性的含义是，规则的前提蕴涵其结论。可靠性的证明采用余归纳（co-induction），直接证明 RGSim 关系的定义成立。我们在证明辅助工具 Coq [29] 中实现了这些规则的可靠性证明 [30]。

依赖/保证条件的实例化　通过以不同方式实例化 RGSim 关系中的依赖条件，我们可以得到第 2.1.1节介绍的串行程序精化关系和第 2.1.2节介绍的假设任意开放环境的精化关系。前者可以用下式 (2.6) 表示。它假设同一的环境，造成 PAR 规则的干涉约束难以被满足。也就是说，串行程序精化关系缺少并发可组合性。这与我们在第 2.1.1节所发现的一致。下式 (2.7) 假设任意的环境，这样 PAR 规则的干涉约束平凡地成立。但是这一假设太强了，实际应用中 (2.7) 式通常无法被满足。

$$(C, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{Id}, \mathsf{True}) \tag{2.6}$$

$$(C, \mathsf{True}, \mathsf{True}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbb{C}, \mathsf{True}, \mathsf{True}) \tag{2.7}$$

2.5　一个简单的例子

下面我们举一个简单的例子来展示 RGSim 及其并发可组合性在程序精化验证中的应用。设程序变换 **T** 将高层程序 $\mathbb{C}_1 ||| \mathbb{C}_2$ 变换为 $C_1 \parallel C_2$，后者用锁 l 来同步对共享变量 x 的访问。我们想要证明 $C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 ||| \mathbb{C}_2$ 成立。也就是说，虽

然 C_2 用两次 x 的增加来实现原子的 x:=x+2 指令，并发的线程 C_1 却不会打印出 x 仅增加了一次时的值。这里我们将输出事件视作外部可观测的行为。

```
print(x);   ||| x := x + 2;
            ⇓
lock(l);      lock(l);
print(x);   ‖ x := x+1; x := x+1;
unlock(l);    ⟨unlock(l); X := x;⟩
```

为了方便证明，我们在低层程序中引入辅助共享变量 X，记录在释放锁的时刻 x 的值。X 描述了 x 在临界区外时的值，它应与高层的 x（在对应操作后）的值相匹配。这里我们用 $\langle C\rangle$ 表示代码 C 是原子执行的，其语义遵照 RGSep [15]（或见第 5.2节）。该辅助变量是只写的（write-only），因此不会改变程序的可观测行为 [9]。下面我们可以仅关注添加辅助代码后的程序。

根据 RGSim 的适当性和并发可组合性，我们只需证明单个线程上的模拟关系在合适的依赖/保证条件下成立。我们首先定义不变关系 α，如下所示，它只关心锁空闲的时候 x 的值。前后条件与 α 相同。

$$\alpha \stackrel{\text{def}}{=} \{(\sigma,\Sigma) \mid \sigma(\mathtt{X}) = \Sigma(\mathtt{x}) \wedge (\sigma(\mathtt{l}) = 0 \Longrightarrow \sigma(\mathtt{x}) = \sigma(\mathtt{X}))\}.$$

高层的线程可以在任意环境下执行，保证任意的状态转换：$\mathbb{R} = \mathbb{G} \stackrel{\text{def}}{=} \mathsf{True}$。低层的线程用锁来保护对 x 的访问，因此低层的依赖/保证条件不是任意的：

$$\begin{aligned}
\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma(\mathtt{l}) = \mathtt{cid} \Longrightarrow \\
\sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{X}) = \sigma'(\mathtt{X}) \wedge \sigma(\mathtt{l}) = \sigma'(\mathtt{l})\}; \\
\mathcal{G} \stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma' = \sigma \ \vee\ \sigma(\mathtt{l}) = 0 \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \\
\vee\ \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{x} \rightsquigarrow _\} \\
\vee\ \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow 0, \mathtt{X} \rightsquigarrow _\}\}.
\end{aligned}$$

每个低层的线程都保证它只会在获得锁之后才修改 x。它的环境不可以在当前线程持有锁时修改 x 或 l。我们用 cid 来表示当前线程的标识号。当某个线程获得锁时，我们将锁置为该线程的标识号。

根据 RGSim 的定义，我们可以证明 $(C_1,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathbb{C}_1,\mathbb{R},\mathbb{G})$ 和 $(C_2,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathbb{C}_2,\mathbb{R},\mathbb{G})$ 成立。应用 PAR 规则后，由 RGSim 的适当性（推论 2.6）可知，对于任何保证 α 的状态变换 $\mathbf{T}$，$C_1 \parallel C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 ||| \mathbb{C}_2$ 成立。

有趣的是，如果我们省略 C_1 中的lock 和unlock 操作，则 $C_1 \parallel C_2$ 会比 $\mathbb{C}_1 ||| \mathbb{C}_2$ 产生更多的外部可观测行为。但这并不意味着我们的 PAR 规则是不可靠的（它是可靠的!）。事实上，在环境的 α-相关的状态转换 $\langle\mathcal{R},\mathbb{R}^*\rangle_\alpha$ 的干扰下，x 可能在低层与高层呈现不同的值，$(\mathtt{print(x)},\mathcal{R},\mathcal{G}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathtt{print(x)},\mathbb{R},\mathbb{G})$ 并不成立（尽管这里低层与高层的代码长得完全一样）。

辅助变量的使用　辅助变量 x 帮助我们定义不变关系 α 并完成证明。若没有它，我们很难证明上述程序精化关系。可能有读者想要直接证明下式：

$$(C_1, \mathcal{R}', \mathcal{G}') \preceq_{\alpha';\alpha'\ltimes\alpha'} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}), \tag{2.8}$$

其中 α'，$\mathcal{R}'$ 和 $\mathcal{G}'$ 是把 α，$\mathcal{R}$ 和 $\mathcal{G}$ 中的 x 删除掉，它们的定义如下：

$$\begin{aligned}
\alpha' &\stackrel{\text{def}}{=} \{(\sigma,\Sigma) \mid \sigma(\mathtt{l})=0 \implies \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}\,;\\
\mathcal{R}' &\stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma(\mathtt{l})=\mathtt{cid} \implies \sigma(\mathtt{x})=\sigma'(\mathtt{x}) \wedge \sigma(\mathtt{l})=\sigma'(\mathtt{l})\}\,;\\
\mathcal{G}' &\stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma'=\sigma \ \vee\ \sigma(\mathtt{l})=0 \wedge \sigma'=\sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\}\\
&\qquad\qquad \vee\ \sigma(\mathtt{l})=\mathtt{cid} \wedge \sigma'=\sigma\{\mathtt{x} \rightsquigarrow _\}\\
&\qquad\qquad \vee\ \sigma(\mathtt{l})=\mathtt{cid} \wedge \sigma'=\sigma\{\mathtt{l} \rightsquigarrow 0\}\}\,.
\end{aligned}$$

然而，(2.8) 式并不成立。原因是 $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ 会允许一些本不该出现的状态转换。例如，对于下述 σ，σ'，Σ 和 Σ'，我们有 $((\sigma,\sigma'),(\Sigma,\Sigma')) \in \langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ 成立：

$$\sigma \ = \ \sigma' \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 0, \mathtt{l} \rightsquigarrow \mathtt{cid}\}\,; \qquad \Sigma \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 0\}\,; \qquad \Sigma' \stackrel{\text{def}}{=} \{\mathtt{x} \rightsquigarrow 1\}\,.$$

此时，即使低层的线程持有锁，高层线程的环境也可以改变 x。那么，C_1 与 $\mathbb{C}_1$ 就可能打印出不同的值，破坏了模拟关系 (2.8) 式。

如果我们以另一种方式定义 RGSim 关系，则可以在验证这个例子时避免使用辅助变量。原本的 RGSim 关系（定义 2.4）在低层和高层使用各自的依赖/保证条件，然后用 α 将它们联系起来。我们可以不这样做，而是直接定义“关系式的依赖/保证条件”：

$$r, g \in \mathscr{P}((\mathit{LState} \times \mathit{LState}) \times (\mathit{HState} \times \mathit{HState}))\,.$$

那么，新的模拟关系就可以用下面的形式，它的定义只是将定义 2.4中出现的所有 $\langle \mathcal{R}, \mathbb{R}^* \rangle_{\alpha}$ 和 $\langle \mathcal{G}, \mathbb{G}^* \rangle_{\alpha}$ 都分别替换为 r 和 g：

$$C \preceq_{r;g;\alpha;\zeta\ltimes\gamma} \mathbb{C}\,.$$

新的模拟关系仍然具有原来的 RGSim 的所有良好性质，包括并发可组合性。但它使得我们不再需要借助辅助变量来证明上面那个简单的例子。我们可以证明 $C_1 \preceq_{\alpha';\alpha'\ltimes\alpha';r;g} \mathbb{C}_1$ 以及 $C_2' \preceq_{\alpha';\alpha'\ltimes\alpha';r;g} \mathbb{C}_2$ 成立，其中 C_2' 是将 C_2 中的 x 去掉后的代码，α' 的定义同上，r 和 g 如下定义：

$$\begin{aligned}
r &\stackrel{\text{def}}{=} \{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid \sigma(\mathtt{l})=\mathtt{cid} \implies\\
&\qquad\qquad \sigma(\mathtt{x})=\sigma'(\mathtt{x}) \wedge \sigma(\mathtt{l})=\sigma'(\mathtt{l}) \wedge \Sigma(\mathtt{x})=\Sigma'(\mathtt{x})\}\,;\\
g &\stackrel{\text{def}}{=} \{((\sigma,\sigma'),(\Sigma,\Sigma')) \mid \sigma'=\sigma \wedge \Sigma'=\Sigma \ \vee\ \sigma(\mathtt{l})=0 \wedge \sigma'=\sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \wedge \Sigma'=\Sigma\\
&\qquad\qquad \vee\ \sigma(\mathtt{l})=\mathtt{cid} \wedge \sigma'=\sigma\{\mathtt{x} \rightsquigarrow _\} \wedge \Sigma'=\Sigma\\
&\qquad\qquad \vee\ \sigma(\mathtt{l})=\mathtt{cid} \wedge \sigma'=\sigma\{\mathtt{l} \rightsquigarrow 0\} \wedge \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x})\}\}\,.
\end{aligned}$$

注意 r 的定义要求：当低层的线程持有锁时，无论高层还是低层的环境都不可以改变 x。在这样的关系式定义的 r 中就不会出现前面讨论的那种本不该出现的状态转换。它比 $\langle \mathcal{R}', \mathbb{R}^* \rangle_{\alpha'}$ 具有更强的表达力，但也可能会显得更加难以理解。

在第 3章和第 4章中，我们会展示 RGSim 的更多应用。我们会使用原本的 RGSim 定义（即定义 2.4），因为对于这两章中的例子，分别定义两层的依赖/保证条件会较为清楚也较为容易。而在第 5章对线性一致性的验证中，定义关系式的 r 和 g 可以使证明更加清晰。在第 5章中，为了方便 r 和 g 的使用，我们还定义了它们的语法形式。

2.6　本章小结

本章提出了一种新颖的模拟关系 RGSim，用以验证并发程序精化。特别地，可以用它来证明并发程序变换的正确性。我们在证明辅助工具 Coq [29] 中实现了 RGSim，证明了它的适当性和可组合性。具体的 Coq 实现可见 [30]。

RGSim 关系保证了低层程序可以保持对应高层程序的安全性（safety）性质，包括部分正确性（partial correctness）。但是，当高层程序终止而低层程序陷入不产生事件的无穷循环（如`while(true) skip;`）时，RGSim 仍然可以被满足。在第 2.5节的例子中，我们允许低层的程序被永远地阻塞（比如当其他某个线程获得了锁却一直不释放时就会出现这种情况）。在并发环境下证明保终止性的精化十分困难，例如可能需要证明没有死锁等，我们将其列为今后的工作之一。

RGSim 的可组合性（图 2.7）允许我们将较大规模的程序上的精化验证分解为程序基本单元上的精化验证，而程序的基本单元往往是原语指令。然而，图 2.7并没有提供原语指令上的相关规则。验证原语指令上的精化时，我们只能使用 RGSim 的语义定义（定义 2.4）。这使得证明过程常常十分繁琐、复杂。在第 5章，我们会针对并发对象的线性一致性验证这一特定应用，设计一套更完整的公理化证明系统。对于通用情况下的完整证明理论，事实上，Turon 等人已经在我们的工作的基础上设计了这样一套逻辑系统 [32]，其基本想法与我们将要在第 5章介绍的程序逻辑相似。

第 3 章　RGSim 的简单应用

上一章的最后我们举了一个简单的例子，展示了如何在程序精化验证中使用 RGSim 及其可组合性。本章展示 RGSim 的更多应用，包括：并发环境下的程序优化验证（第 3.1节），以及抽象算法和并发对象的细粒度实现的验证（第 3.2节）。

3.1　并发环境下程序优化的验证

作为程序精化验证的一个直接应用，编译优化验证要求证明目标程序是对源程序的精化。RGSim 关系让我们可以验证并发环境下的编译优化。下面我们应用 RGSim 将 Benton 在串行环境下的优化验证工作 [27] 移植到并发环境下。

3.1.1　专为程序优化设计的推理规则

优化往往对上下文有特定的要求。例如，删除赋值语句 $x := E$ 的条件是其后的代码不会引用 x（即它是无用代码）。对于并发程序的优化，我们需要考虑并发的上下文。RGSim 关系让我们用依赖/保证条件描述优化对并发上下文的要求。下面我们基于 RGSim 关系给出一些推理规则，用以刻画及证明在特定并发上下文中的常用程序优化（如死代码删除等）的正确性。注意这里源程序和目标程序使用同一编程语言。

SEQUENTIAL-UNIT 规则

$$\frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(\textbf{skip}; C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)} \qquad \frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{skip}; C_2, \mathcal{R}_2, \mathcal{G}_2)}$$

此外还有两条规则，用以在 C_1 或 C_2 之后顺序组合空语句 **skip**。也就是说，**skip** 语句可以被任意地引入或删除。

COMMON-BRANCH 规则

$$\frac{\begin{array}{c}\forall \sigma_1, \sigma_2.\ (\sigma_1, \sigma_2) \in \zeta \implies B\ \sigma_2 \neq \bot \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1\ltimes\gamma} (C_1, \mathcal{R}', \mathcal{G}') \qquad \zeta_1 = (\zeta \cap (\textbf{true} \Lambda\!\!\Lambda B)) \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2\ltimes\gamma} (C_2, \mathcal{R}', \mathcal{G}') \qquad \zeta_2 = (\zeta \cap (\textbf{true} \Lambda\!\!\Lambda \neg B))\end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2, \mathcal{R}', \mathcal{G}')}$$

如果 if 语句的条件总是可以求值，且它的两个分支都可以优化为同一代码 C，那么我们就可以将整个 if 语句优化为代码 C。

KNOWN-BRANCH 规则

$$\frac{(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (C_1,\mathcal{R}',\mathcal{G}') \qquad \zeta = (\zeta \cap (\mathbf{true} \Cap B))}{(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{if}\,(B)\,C_1\ \mathbf{else}\,C_2,\mathcal{R}',\mathcal{G}')}$$

$$\frac{(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (C_2,\mathcal{R}',\mathcal{G}') \qquad \zeta = (\zeta \cap (\mathbf{true} \Cap \neg B))}{(C,\mathcal{R},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{if}\,(B)\,C_1\ \mathbf{else}\,C_2,\mathcal{R}',\mathcal{G}')}$$

如果 if 语句的条件 B 求值为 **true**（或 **false**），那么我们就只需考虑它的 then 分支（或 else 分支）。这两条规则可以由 COMMON-BRANCH 规则推导得到。

DEAD-WHILE 规则

$$\frac{\zeta = (\zeta \cap (\mathbf{true} \Cap \neg B)) \qquad \zeta \subseteq \alpha \qquad \mathsf{Sta}(\zeta,\langle\mathcal{R}_1,\mathcal{R}_2^*\rangle_\alpha)}{(\mathbf{skip},\mathcal{R}_1,\mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\zeta} (\mathbf{while}\,(B)\{C\},\mathcal{R}_2,\mathsf{Id})}$$

当循环条件求值为 **false** 时，我们可以删除整个循环。注意在并发环境下，前条件 ζ 应当是稳定的。也就是说，无论环境如何影响，循环条件总是求值为 **false**。

LOOP-PEELING 规则

$$\frac{(\mathbf{while}\,(B)\{C\},\mathcal{R}_1,\mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\,(B)\{C\},\mathcal{R}_2,\mathcal{G}_2)}{(\mathbf{if}\,(B)\,\{C;\mathbf{while}\,(B)\{C\}\}\ \mathbf{else\ skip},\mathcal{R}_1,\mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\,(B)\{C\},\mathcal{R}_2,\mathcal{G}_2)}$$

LOOP-UNROLLING 规则

$$\frac{(\mathbf{while}\,(B)\{C\},\mathcal{R}_1,\mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\,(B)\{C\},\mathcal{R}_2,\mathcal{G}_2)}{(\mathbf{while}\,(B)\{C;\mathbf{if}\,(B)\,C\ \mathbf{else\ skip}\},\mathcal{R}_1,\mathcal{G}_1) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{while}\,(B)\{C\},\mathcal{R}_2,\mathcal{G}_2)}$$

DEAD-CODE-ELIMINATION 规则

$$\frac{(\mathbf{skip},\mathsf{Id},\mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C,\mathsf{Id},\mathcal{G}) \qquad \mathsf{Sta}(\{\zeta,\gamma\},\langle\mathcal{R}_1,\mathcal{R}_2^*\rangle_\alpha)}{(\mathbf{skip},\mathcal{R}_1,\mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C,\mathcal{R}_2,\mathcal{G})}$$

直观上，$(\mathbf{skip},\mathsf{Id},\mathsf{Id}) \preceq_{\alpha;\zeta\ltimes\gamma} (C,\mathsf{Id},\mathcal{G})$ 是指在串行环境下删除代码 C，其中初始和终止状态分别满足 ζ 和 γ。若 ζ 和 γ 在环境 $\mathcal{R}_1$ 和 $\mathcal{R}_2$ 的影响下是稳定的，则我们也可以在这样的并发环境下删除代码 C。

REDUNDANCY-INTRODUCTION 规则

$$\frac{(c,\mathsf{Id},\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{skip},\mathsf{Id},\mathsf{Id}) \qquad \mathsf{Sta}(\{\zeta,\gamma\},\langle\mathcal{R}_1,\mathcal{R}_2^*\rangle_\alpha)}{(c,\mathcal{R}_1,\mathcal{G}) \preceq_{\alpha;\zeta\ltimes\gamma} (\mathbf{skip},\mathcal{R}_2,\mathsf{Id})}$$

类似上一条规则将串行环境下的死代码删除优化移植到并发环境下，我们也可以将串行环境下的冗余代码引入优化移植到并发环境下。这里同样要求前后条件在并发环境下是稳定的。注意引入的冗余代码 c 是一条原语指令。当引入一串冗余代码时，我们必须考虑环境对代码执行过程的每个中间状态的影响。

使用上述这些规则，我们可以证明许多传统的编译优化在特定上下文中应用在并发程序上时也是正确的。下面我们举几个例子：循环不变代码外提，强度削弱和归纳变量删除。

3.1.2　应用举例一：循环不变代码外提

我们首先形式化地证明第 2.1.5节中的例子。如前所述，外提循环不变代码 `t:=x+1` 的条件是：环境 $\mathcal{R}$ 不能修改 `x` 或 `t`。

$$\mathcal{R} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{t}) = \sigma'(\mathtt{t})\}.$$

程序的“保证条件”可以包含任意的状态转换。不变条件 α 只关心 `i`，`n` 和 `x` 的值。前后条件与 α 相同。

$$\alpha \stackrel{\text{def}}{=} \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \wedge \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}.$$

那么，这一优化的正确性可以用如下 RGSim 关系表示：

$$(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C, \mathcal{R}, \mathsf{True}). \tag{3.1}$$

我们可以用 RGSim 的定义和代码的操作语义直接证明 (3.1) 式。下面我们采取另一种更便捷的办法，我们应用 RGSim 的可组合性规则和上一节介绍的优化规则来证明 (3.1) 式。首先，应用 DEAD-CODE-ELIMINATION 规则和 REDUNDANCY-INTRODUCTION 规则，可得到下述 RGSim 关系：

$$\begin{aligned}&(\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\gamma} (\textbf{skip}, \mathcal{R}, \mathsf{True});\\&(\textbf{skip}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\gamma\ltimes\eta} (\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True}),\end{aligned}$$

其中 γ 和 η 描述了特定程序点的状态：

$$\begin{aligned}\gamma &\stackrel{\text{def}}{=} \alpha \cap \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{t}) = \sigma_1(\mathtt{x}) + 1\};\\ \eta &\stackrel{\text{def}}{=} \gamma \cap \{(\sigma_1, \sigma) \mid \sigma(\mathtt{t}) = \sigma(\mathtt{x}) + 1\}.\end{aligned}$$

然后，应用可组合性规则 SEQ 和 WHILE，我们得到 $(C_1', \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C', \mathcal{R}, \mathsf{True})$，其中 C_1' 和 C' 分别在 C_1 和 C 的基础上增加 **skip** 语句：

```
C1' :                          C' :
t := x + 1;                    skip;
while(i < n) {                 while(i < n) {
  skip;                          t := x + 1;
  i := i + t;                    i := i + t;
}                              }
```

此外，应用 SEQUENTIAL-UNIT 规则和可组合性规则 SEQ 和 WHILE，可得 $(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C_1', \mathcal{R}, \mathsf{True})$ 和 $(C', \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C, \mathcal{R}, \mathsf{True})$。最后，应用 TRANS 规则，我们就得到了 (3.1) 式。也就是说，这一优化在特定的并发上下文中依然是正确的。这里 $\mathcal{R}$ 不允许修改 `x` 和 `t`，但可以修改 `i` 和 `n`，或者读 `x`。与这样的线程一起并发执行时，C_1 仍然是对 C 的正确的优化。

3.1.3　应用举例二：强度削弱和归纳变量删除

目标程序C_2

```
local k, r;
k := 0;
r := 6*n;
while(k<r) {
  x := x+k;
  k := k+6;
}
```

⇐

中间程序C_1

```
local i, k;
i := 0;
k := 0;
while(i<n) {
  x := x+k;
  i := i+1;
  k := k+6;
}
```

⇐

源程序C

```
local i;
i := 0;
while(i<n) {
  x := x+6*i;
  i := i+1;
}
```

在这个例子中，我们先对源程序 C 做强度削弱，引入局部变量 k 并且用加法代替乘法，得到中间程序 C_1。环境不可以修改归纳变量 i 和新引入的局部变量 k。然后，我们删除 i，将中间程序 C_1 变换为目标程序 C_2。后者使用 k 作为新的归纳变量。我们假设目标环境不会修改 n 和 r，这样我们就可以计算出 k 的边界，作为新的循环条件。下面我们给出源程序、中间程序和目标程序的环境 $\mathcal{R}$、$\mathcal{R}_1$ 和 $\mathcal{R}_2$：

$$
\begin{aligned}
\mathcal{R} &\stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma(\mathtt{i}) = \sigma'(\mathtt{i})\} \\
\mathcal{R}_1 &\stackrel{\text{def}}{=} \{(\sigma_1,\sigma_1') \mid \sigma_1(\mathtt{i}) = \sigma_1'(\mathtt{i}) \wedge \sigma_1(\mathtt{k}) = \sigma_1'(\mathtt{k})\} \\
\mathcal{R}_2 &\stackrel{\text{def}}{=} \{(\sigma_2,\sigma_2') \mid \sigma_2(\mathtt{k}) = \sigma_2'(\mathtt{k}) \wedge \sigma_2(\mathtt{r}) = \sigma_2'(\mathtt{r}) \wedge \sigma_2(\mathtt{n}) = \sigma_2'(\mathtt{n})\}
\end{aligned}
$$

不变条件应要求两个程序用到的公共变量具有相同的值。因此，C 和 C_1 之间的不变条件 α，以及 C_1 和 C_2 之间的不变条件 β 如下定义：

$$
\begin{aligned}
\alpha &\stackrel{\text{def}}{=} \{(\sigma_1,\sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \wedge \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}\,; \\
\beta &\stackrel{\text{def}}{=} \{(\sigma_2,\sigma_1) \mid \sigma_2(\mathtt{k}) = \sigma_1(\mathtt{k}) \wedge \sigma_2(\mathtt{n}) = \sigma_1(\mathtt{n}) \wedge \sigma_2(\mathtt{x}) = \sigma_1(\mathtt{x})\}\,.
\end{aligned}
$$

那么，两段程序优化的正确性可以如下表示：

$$
(C_2,\mathcal{R}_2,\mathsf{True}) \preceq_{\beta;\beta\ltimes\beta} (C_1,\mathcal{R}_1,\mathsf{True})\,, \quad (C_1,\mathcal{R}_1,\mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (C,\mathcal{R},\mathsf{True})\,.
$$

我们可以用 RGSim 的定义直接证明它们，也可以应用优化规则（如 DEAD-CODE-ELIMINATION 和 REDUNDANCY-INTRODUCTION 规则）和可组合性规则证明它们。证明过程类似上一个例子，在此省略。

之后，我们可以用 TRANS 规则将两段优化的验证组合起来，得到：

$$
(C_2,\mathcal{R}_2,\mathsf{True}) \preceq_{\alpha\circ\beta;\alpha\circ\beta\ltimes\alpha\circ\beta} (C,\mathcal{R},\mathsf{True})\,,
$$

其中 $\alpha\circ\beta = \{(\sigma_2,\sigma) \mid \sigma_2(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_2(\mathtt{x}) = \sigma(\mathtt{x})\}$。这说明，当源程序的环境不改变 i 和 n 时，我们就可以应用强度削弱和归纳变量删除，将源程序 C 变换为目标程序 C_2。

3.1.4　相关工作

我们用 RGSim 验证了并发环境下程序优化的正确性，这部分工作仿照了 Benton 对串行程序优化的验证工作 [27]，后者同样是以推理规则的形式来说明优化的正确性。此外，RGSim 在顺序组合、条件语句以及循环上的可组合性规则均与 Benton 的关系式霍尔逻辑 [27] 以及 Yang 的关系式分离逻辑 [33] 一致。我们尚未将 RGSim 应用于验证现实世界中的并发程序编译器，或应用于在并发环境下验证更复杂的优化算法，如懒惰代码移动（lazy code motion）等。我们将在今后推进这方面的研究工作。

并发程序语言的编译器验证工作可以追溯至上个世纪九十年代对带有消息传递机制的函数式语言的编译器验证 [34, 35]。最近，Lochbihler 提出了一个针对 Java 线程的编译器并用弱互模拟关系（weak bisimulation）验证其正确性 [36]。他把每个堆更新操作都视为可观测的，因此不允许目标程序与源程序具有不同粒度的原子更新操作。为了获得并发可组合性，他要求在任何共享状态的状态转换下模拟关系都能被保持，也就是说，他假设任意的并发环境。而正如我们在第 2.1.2节中解释的那样，这一要求对于许多精化应用（包括本节中的例子）都太强了。

Burckhardt 等人提出一种在弱内存模型（weak memory model）下验证并发程序变换的方法 [37]。他们的方法依赖一种基于程序路径的指称语义（denotational semantics），虽然具有可组合性，但在任何程序点共享变量的值都被视作任意的。也就是说，他们也假设任意的并发环境，缺乏实用性。

在 Leroy 的 CompCert 项目 [1] 的基础上，Ševčík 等人用模拟关系验证了一个并发程序编译器 [28]，其源语言是一种类 C 的并发语言，目标语言是 x86。然而，他们的编译器中有两个阶段的证明不具有可组合性。

我们提出的 RGSim 是一种通用的、具有可组合性的并发程序精化验证技术。对并发环境下程序优化的验证仅仅是它的众多应用之一。我们会在下一节和下一章展示 RGSim 的更多应用。

3.2　抽象操作的细粒度实现的验证

前面已经提到，算法或程序的验证可以归结为证明可执行的具体程序是对相应的抽象操作的精化。在并发环境下，我们可以用 RGSim 证明抽象算法的细粒度实现的正确性。下面我们以一个并发计算 GCD（最大公约数）的程序 [24] 为例，展示 RGSim 在并发程序正确性验证方面的应用。

A_1 :

```
 1  local d1;
 2  d1 := 0;
 3  while (d1 = 0) {
 4    atom{
 5      if (a = b)
 6        d1 := 1;
 7      if (a > b)
 8        a := a - b;
 9    }
10  }
```

$|||$

A_2 :

```
 1  local d2;
 2  d2 := 0;
 3  while (d2 = 0) {
 4    atom{
 5      if (b = a)
 6        d2 := 1;
 7      if (b > a)
 8        b := b - a;
 9    }
10  }
```

(a) 抽象程序

C_1 :

```
 1  local d1, t11, t12;
 2  d1 := 0;
 3  while (d1 = 0) {
 4    t11 := a;
 5    t12 := b;
 6    if (t11 = t12)
 7      d1 := 1;
 8    if (t11 > t12)
 9      a := t11 - t12;
10  }
```

$\|$

C_2 :

```
 1  local d2, t21, t22;
 2  d2 := 0;
 3  while (d2 = 0) {
 4    t21 := b;
 5    t22 := a;
 6    if (t21 = t22)
 7      d2 := 1;
 8    if (t21 > t22)
 9      b := t21 - t22;
10  }
```

(b) 具体实现

图 3.1 计算最大公约数的并发程序

3.2.1　应用举例：计算最大公约数的并发程序

图 3.1(b) 展示了并发计算 GCD 的具体实现代码。该程序使用两个线程，计算共享变量 a 和 b 的最大公约数。一个线程执行 C_1，读取 a 和 b 的值，并在 a > b 时更新 a。另一个线程执行相反的操作 C_2。当 a = b 时，两个线程均终止。该代码实现了图 3.1(a) 中的更为抽象的程序，后者中的两个线程分别原子地改变 a 和 b。这里我们用原子块 ***atom***$\{\mathbb{C}\}$ 表示 $\mathbb{C}$ 是原子执行的。

我们想要证明具体的和抽象的 GCD 程序总是计算出同样的结果，即 $(C_1 \| C_2);\texttt{print(a)}$ 和 $(A_1 ||| A_2);\texttt{print(a)}$ 有相同的输出。这里 print(a) 可以打印计算结果。

根据 RGSim 的适当性和可组合性，我们只需证明 C_1 和 A_1（以及 C_2 和 A_2）中的核心的更新操作是等价的，即证明 C_1 的第 4 行至第 9 行代码（记作 C_1'）与 A_1 中第 4 行至第 9 行的原子块（记作 A_1'）是等价的（以及 C_2 和 A_2 的第 4 行至第 9 行等价）。

我们首先定义 α 关系，要求低层与高层共有的变量具有相同的值：

$$\alpha \stackrel{\text{def}}{=} \{(\sigma,\Sigma) \mid \sigma(\mathtt{a}) = \Sigma(\mathtt{a}) \wedge \sigma(\mathtt{b}) = \Sigma(\mathtt{b}) \wedge \sigma(\mathtt{d1}) = \Sigma(\mathtt{d1}) \wedge \sigma(\mathtt{d2}) = \Sigma(\mathtt{d2})\}.$$

线程的依赖/保证条件可以如下定义，其中一个线程的依赖条件就是另一个线程的保证条件：

$$\begin{aligned}
\mathcal{R}_1 = \mathcal{G}_2 &\stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma'(\mathtt{t11}) = \sigma(\mathtt{t11}) \wedge \sigma'(\mathtt{t12}) = \sigma(\mathtt{t12}) \\
&\qquad \wedge \sigma'(\mathtt{d1}) = \sigma(\mathtt{d1}) \wedge \sigma'(\mathtt{a}) = \sigma(\mathtt{a}) \wedge (\sigma(\mathtt{a}) \geq \sigma(\mathtt{b}) \Rightarrow \sigma'(\mathtt{b}) = \sigma(\mathtt{b}))\} \\
\mathcal{R}_2 = \mathcal{G}_1 &\stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid \sigma'(\mathtt{t21}) = \sigma(\mathtt{t21}) \wedge \sigma'(\mathtt{t22}) = \sigma(\mathtt{t22}) \\
&\qquad \wedge \sigma'(\mathtt{d2}) = \sigma(\mathtt{d2}) \wedge \sigma'(\mathtt{b}) = \sigma(\mathtt{b}) \wedge (\sigma(\mathtt{b}) \geq \sigma(\mathtt{a}) \Rightarrow \sigma'(\mathtt{a}) = \sigma(\mathtt{a}))\} \\
\mathbb{R}_1 = \mathbb{G}_2 &\stackrel{\text{def}}{=} \{(\Sigma,\Sigma') \mid \Sigma'(\mathtt{d1}) = \Sigma(\mathtt{d1}) \wedge \Sigma'(\mathtt{a}) = \Sigma(\mathtt{a}) \wedge (\Sigma(\mathtt{a}) \geq \Sigma(\mathtt{b}) \Rightarrow \Sigma'(\mathtt{b}) = \Sigma(\mathtt{b}))\} \\
\mathbb{R}_2 = \mathbb{G}_1 &\stackrel{\text{def}}{=} \{(\Sigma,\Sigma') \mid \Sigma'(\mathtt{d2}) = \Sigma(\mathtt{d2}) \wedge \Sigma'(\mathtt{b}) = \Sigma(\mathtt{b}) \wedge (\Sigma(\mathtt{b}) \geq \Sigma(\mathtt{a}) \Rightarrow \Sigma'(\mathtt{a}) = \Sigma(\mathtt{a}))\}
\end{aligned}$$

然后，根据操作语义，我们可以证明 C_1' 和 A_1' 之间的（两个方向的）RGSim 关系成立，如下所示。这里 α^{-1} 是 α 的逆关系，在图 2.6中定义。

$$(C_1', \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha\ltimes\alpha} (A_1', \mathbb{R}_1, \mathbb{G}_1)\,, \qquad (A_1', \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_1', \mathcal{R}_1, \mathcal{G}_1).$$

根据 WHILE 和 SEQ 规则，我们可以得到 C_1 和 A_2 之间的 RGSim 关系：

$$(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\alpha\ltimes\alpha} (A_1, \mathbb{R}_1, \mathbb{G}_1)\,, \qquad (A_1, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_1, \mathcal{R}_1, \mathcal{G}_1).$$

类似地，C_2 和 A_2 之间的 RGSim 关系也成立：

$$(C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\alpha\ltimes\alpha} (A_2, \mathbb{R}_2, \mathbb{G}_2)\,, \qquad (A_2, \mathbb{R}_1, \mathbb{G}_1) \preceq_{\alpha^{-1};\alpha^{-1}\ltimes\alpha^{-1}} (C_2, \mathcal{R}_1, \mathcal{G}_1).$$

当 C_1 和 C_2（或 A_1 和 A_2）并发执行时，整个 GCD 计算程序依赖的环境不可以修改共享变量 a 和 b，因此我们令环境为 Id。整个程序的保证条件可以为 True，包含任意的状态转换。我们可以证明 $(\mathtt{print(a)}, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} (\mathtt{print(a)}, \mathsf{Id}, \mathsf{True})$ 及反方向的关系都成立。那么，由 PAR 和 SEQ 规则可得：

$$((C_1 \parallel C_2); \mathtt{print(a)}, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\alpha\ltimes\alpha} ((A_1 \,|||\, A_2); \mathtt{print(a)}, \mathsf{Id}, \mathsf{True})\,,$$

且反方向的关系也成立。根据 RGSim 的适当性（推论 2.6）可知，对于任何满足 α 的状态变换 $\mathbf{T}$，下述结论成立：

$$(C_1 \parallel C_2); \mathtt{print(a)} \approx_{\mathbf{T}} (A_1 \,|||\, A_2); \mathtt{print(a)}\,.$$

至此我们证明了图 3.1中的细粒度的具体实现和粗粒度的抽象程序总是可以计算出同样的结果。不难看出抽象程序确实计算了 a 和 b 的 GCD，因此我们知道具体实现也正确计算了 GCD。这个例子说明，验证一个复杂程序时，可以先证明它与一个较简单的程序等价，然后验证那个简单的程序。

3.2.2　并发对象的验证

并发对象封装了共享数据结构和操作该数据结构的方法。多线程的客户端程序只能通过调用这些方法来访问并发对象。对象实现的功能正确性通常定义为线性一致性（linearizability）[4]，它要求对象实现具有原子的行为——每个方法操作都应在调用和返回之间的每个时刻“起作用”（take effect）。我们可以将对象的规范定义为高层抽象的原子操作，那么，线性一致性就建立了在特定的并发环境下，细粒度的对象具体实现和对应的抽象原子操作之间的精化关系。

我们应用 RGSim 证明了许多并发对象的线性一致性，如非阻塞型并发计数器 [20]，Treiber 的栈算法 [10]，锁耦合（lock-coupling）的链表实现 [3]，等等。然而，RGSim 并不支持可线性化点不固定的并发对象，包括使用帮助机制实现的对象（如 HSY 栈算法 [18]），可线性化点依赖于未来的对象（如懒惰集合算法 [19]）以及同时涉及这两种特性的对象（如 RDCSS 算法 [15]）。正如我们在第 1.1.2节中所介绍的，这些并发对象的验证难度很大。我们将在第 5章扩展 RGSim 关系，并提出一种程序逻辑来验证这些可线性化点不固定的并发对象。该程序逻辑和新的模拟关系同样也能验证 RGSim 所支持的简单对象。由于第 5章中的验证方法更高效、更直观，我们就不在此展示应用 RGSim 验证的例子。感兴趣的读者请参考第 5章，我们会详细地、形式化地介绍了线性一致性的含义和验证方法，并给出一些有趣的验证实例。

下一章我们将介绍 RGSim 的另一个重要应用——并发垃圾收集算法的验证。

第 4 章　并发垃圾收集算法的验证

本章我们详细介绍如何将并发垃圾收集（garbage collection，简写为 GC）算法的验证归约为程序精化验证（第 4.1节）。然后我们基于 RGSim 设计出一个通用的 GC 验证框架（第 4.2节）。我们应用这一框架证明 Boehm 等人提出的并发 GC 算法 [23] 的正确性（第 4.3节）。

4.1　并发垃圾收集算法的正确性

并发垃圾收集器由一个专门的线程运行，它与多线程的用户程序（mutator）并发执行。用户线程通过读、写、分配操作来访问共享内存堆（heap）。为了保证 GC 线程与用户线程对堆的视图是一致的，往往需要在用户线程的堆操作中插入额外的代码来与 GC 交互、协作。这些插入额外代码后的堆操作被称作拦截器（barrier），如，读拦截器、写拦截器等。

并发 GC 算法包括 GC 线程与配套的拦截器。它为用户程序员提供了抽象的、友好的编程模型，程序员使用高层的带垃圾收集机制的编程语言（如 Java），用常规内存操作来访问堆，而且不再需要手工地释放无用对象。程序员不需要了解 GC 的实现细节，也不需要知道拦截器的存在。

我们可以使用霍尔风格的程序逻辑验证 GC 线程和拦截器满足它们的程序规范。可是，我们不知道这样验证后的 GC 算法能否为高层的程序员提供正确的抽象，能否保证实际执行的用户线程的行为保持其高层的抽象行为。通常这一部分（即程序规范能够提供这些保证）是根据经验判断并信任的。

这里我们提出一种更为直接的验证方法。我们将并发 GC 算法视作一种程序变换 **T**，将高层的带 GC 机制的编程语言的程序翻译为低层的可执行的程序。高层的原子的内存操作被翻译为低层的对应的拦截器代码。我们假设高层有一个*抽象 GC 线程*，它把不可达的内存对象变成可再分配的内存。该抽象 GC 线程 *AbsGC* 被翻译为低层的实际 GC 代码 C_{gc}，与低层的用户线程并发执行。也就是说，**T** 可以如下定义：

$$\mathbf{T}(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n) \stackrel{\mathrm{def}}{=} \mathsf{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \,||\, \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \,||\, \ldots \,||\, \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n)\,,$$

其中 $\mathbf{T}(\mathbb{C})$ 将 $\mathbb{C}$ 中的内存访问指令变换为对应的拦截器代码，并保持其他代码不变。这里我们引入抽象 GC 的原因是我们假设高层的内存有限。低层与实际机

器对应，其内存是有限的；若我们假设高层具有无限的内存，则两层内存间的双射的定义会变得复杂。

将并发 GC 算法视作上述程序变换 **T** 后，并发 GC 算法的正确性就可以归结为 Correct(**T**)，它要求任何用户程序的高层行为都能在使用该并发 GC 算法时被保持。

4.2　基于 RGSim 的通用验证框架

借助 RGSim 的可组合性，我们可以得到一个证明上述 Correct(**T**) 的通用框架。由 RGSim 的并发可组合性（图 2.7中的 PAR 规则），我们将精化证明分解为 GC 线程和各个用户线程上的证明。对于每个用户线程，我们进一步应用 RGSim 的可组合性（图 2.7 中的 SEQ，IF 和 WHILE 规则），将精化证明分解为每条原语指令上的证明。注意若不借助可组合性，则我们很难进行上述这些证明分解并得到类似的验证框架。

GC 线程的验证　抽象 GC 线程的操作语义可以用一个二元状态谓词 AbsGCStep 来定义：

$$\frac{(\Sigma, \Sigma') \in \mathsf{AbsGCStep}}{(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC}, \Sigma) \longrightarrow (\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC}, \Sigma')}$$

抽象 GC 线程总是以 AbsGCStep 的方式改变高层的状态。对于不同的 GC 算法，我们可以选择不同的 AbsGCStep。通常 AbsGCStep 保证不会修改堆中的可达对象。

验证 GC 线程，就是要证明在并发环境下 C_{gc} 是对 *AbsGC* 的精化。实际上，这可以归约成用传统的 Rely-Guarantee 逻辑 [22] 验证 C_{gc} 满足适当的程序规范，即证明 $\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \{p_{\mathrm{gc}}\} C_{\mathrm{gc}} \{q_{\mathrm{gc}}\}$ 成立，同时要求这里的 $\mathcal{G}_{\mathrm{gc}}$ 是 AbsGCStep 的具体表示。逻辑判断（judgment）$\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \{p_{\mathrm{gc}}\} C_{\mathrm{gc}} \{q_{\mathrm{gc}}\}$ 的含义是，若初始状态满足前条件 p_{gc}，且环境的行为满足 $\mathcal{R}_{\mathrm{gc}}$，则 C_{gc} 的每一步满足 $\mathcal{G}_{\mathrm{gc}}$，且当它终止时后条件 q_{gc} 成立。一般来讲，GC 永远不会终止，因此 q_{gc} 可以是 **false**。$\mathcal{G}_{\mathrm{gc}}$ 和 p_{gc} 由验证程序的人提供，其中 p_{gc} 应当能被任何可能的低层初始状态所满足。$\mathcal{R}_{\mathrm{gc}}$ 包含用户线程的可能行为，它是可以推导得到的，我们会在下面介绍。

用户线程的验证　由于 **T** 在每个用户线程 $\mathbb{C}$ 上的变换是语法制导的（syntax-directed），我们可以应用 RGSim 的可组合性，将 $\mathbb{C}$ 上的精化验证归约为各个原语指令上的精化验证。首先我们需要定义合适的依赖/保证条件。设 $\mathbb{G}^{\mathsf{t}}_{\mathbb{C}}$ 和 $\mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{C})}$

分别表示用户线程 t 的高层指令 $\mathbb{c}$ 和对应的低层实现代码 $\mathbf{T}(\mathbb{c})$ 的保证条件，则线程 t 的保证条件为：

$$\mathcal{G}(\mathsf{t}) \stackrel{\text{def}}{=} \bigcup_{\mathbb{c}} \mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{c})}\,; \qquad \mathbb{G}(\mathsf{t}) \stackrel{\text{def}}{=} \bigcup_{\mathbb{c}} \mathbb{G}^{\mathsf{t}}_{\mathbb{c}}\,. \tag{4.1}$$

线程 t 的依赖条件应包括其他所有用户线程的保证条件，以及 GC 线程的行为：

$$\mathcal{R}(\mathsf{t}) \stackrel{\text{def}}{=} \mathcal{G}_{\text{gc}} \cup (\bigcup_{\mathsf{t}'\neq\mathsf{t}} \mathcal{G}(\mathsf{t}'))\,; \qquad \mathbb{R}(\mathsf{t}) \stackrel{\text{def}}{=} \mathsf{AbsGCStep} \cup (\bigcup_{\mathsf{t}'\neq\mathsf{t}} \mathbb{G}(\mathsf{t}'))\,. \tag{4.2}$$

而前面提到的验证 GC 代码时需要的 $\mathcal{R}_{\text{gc}}$ 就可以如下定义：

$$\mathcal{R}_{\text{gc}} \stackrel{\text{def}}{=} \bigcup_{\mathsf{t}} \mathcal{G}(\mathsf{t})\,. \tag{4.3}$$

用 RGSim 验证精化还需要定义二元关系 α，ζ 和 γ。不变条件 α 将低层状态 σ 与高层状态 Σ 联系起来，且应在每个低层程序步下被保持。一般来讲，α 会在低层给 GC 线程一个具体的局部存储区（store），在堆中增加额外的结构（从而为垃圾收集记录信息），重命名堆对象（如拷贝收集算法），等等。前后条件 ζ 和 γ 以线程标识号 t 为参数。对于每个用户线程 t，$\zeta(\mathsf{t})$ 和 $\gamma(\mathsf{t})$ 应分别在程序变换 $\mathbf{T}$ 的基本单元（即高层原语指令）之前和之后成立。为了支持指令的顺序组合，我们令 $\gamma(\mathsf{t})$ 与 $\zeta(\mathsf{t})$ 一样。我们要求 $\mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t}))$（图 2.6中定义）成立，即 $\zeta(\mathsf{t})$ 应在程序的初始状态上成立。此外，低层和高层对应的布尔表达式应在 ζ-相关的状态下求值相同，这是图 2.7中的 IF 和 WHILE 规则要求的。因此，我们定义：

$$\mathsf{Good}_{\mathbf{T}}(\zeta(\mathsf{t})) \stackrel{\text{def}}{=} \mathsf{InitRel}_{\mathbf{T}}(\zeta(\mathsf{t})) \wedge \forall\mathbb{B}.\ \zeta(\mathsf{t}) \subseteq (\mathbf{T}(\mathbb{B}) \Leftrightarrow \mathbb{B})\,. \tag{4.4}$$

定理 4.1 (并发 GC 验证框架)**.** *设对于任何 $\mathbb{c}$ 和 t 存在 $\mathbb{G}^{\mathsf{t}}_{\mathbb{c}}$，$\mathcal{G}^{\mathsf{t}}_{\mathbf{T}(\mathbb{c})}$，$\zeta(\mathsf{t})$，$\alpha$，$\mathcal{G}_{\text{gc}}$ 和 p_{gc} 使得下述成立（其中 $\mathcal{G}(\mathsf{t})$，$\mathbb{G}(\mathsf{t})$，$\mathcal{R}(\mathsf{t})$，$\mathbb{R}(\mathsf{t})$ 和 $\mathcal{R}_{\text{gc}}$ 分别定义在 (4.1) 式，(4.2) 式和 (4.3) 式中，且 (4.4) 式定义的 $\mathsf{Good}_{\mathbf{T}}(\zeta(\mathsf{t}))$ 成立）：*

1. *（$\mathbf{T}$ 在用户指令上变换的正确性）*
   $\forall \mathsf{t}, \mathbb{c}.\ (\mathbf{T}(\mathbb{c}), \mathcal{R}(\mathsf{t}), \mathcal{G}(\mathsf{t})) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (\mathbb{c}, \mathbb{R}(\mathsf{t}), \mathbb{G}(\mathsf{t}))$；
2. *（GC 代码的验证）*
   $\mathcal{R}_{\text{gc}}; \mathcal{G}_{\text{gc}} \vdash \{p_{\text{gc}}\} C_{\text{gc}} \{\mathbf{false}\}$；
3. *（附加条件）*
   $\mathcal{G}_{\text{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*$；且 $\forall \sigma, \Sigma.\ \sigma = \mathbf{T}(\Sigma) \implies p_{\text{gc}}\ \sigma$；

那么 $\mathsf{Correct}(\mathbf{T})$ 成立。

也就是说，验证并发 GC 时，我们需要做下列事情：

- 定义 α 和 $\zeta(\mathsf{t})$ 关系，并证明 $\mathbf{T}$ 在高层原语指令上变换的正确性。$\mathbf{T}$ 在大部分指令上的变换维持其语法不变，而当低层与高层的指令一样时，它们之

间的模拟关系证明往往是比较直接的。但是对于变换为拦截器代码的高层指令，我们需要证明拦截器不仅实现了高层的指令（满足 RGSim），而且保证了与 GC 的交互机制（满足特定的保证条件）。

- 找到合适的 $\mathcal{G}_{\text{gc}}$ 和 p_{gc}，并用 Rely-Guarantee 逻辑验证 GC 代码。我们要求 GC 的保证条件 $\mathcal{G}_{\text{gc}}$ 不可以比 AbsGCStep 具有更多的行为（见第一个附加条件）。而且，对于任何由某个高层状态经过 $\mathbf{T}$ 变换得到的低层状态 σ，C_{gc} 都可以从 σ 开始执行（见第二个附加条件）。

证明. 为了证明定理 4.1，我们首先由定理假设 2 和 3 证明：

$$(C_{\text{gc}}, \mathcal{R}_{\text{gc}}, \mathcal{G}_{\text{gc}}) \preceq_{\alpha;\zeta_{\text{gc}}\ltimes\zeta_{\text{gc}}} (\mathit{AbsGC}, \mathsf{True}, \mathsf{AbsGCStep})$$

其中 $\zeta_{\text{gc}} \stackrel{\text{def}}{=} \{(\sigma, \Sigma) \mid \sigma = \mathbf{T}(\Sigma)\}$。该证明过程直接遵照 RGSim 的定义。

然后，由定理假设 1 和 RGSim 的可组合性，对程序结构作归纳，可得：

$$\begin{aligned}&\forall \mathbb{C}_1, \ldots, \mathbb{C}_n.\ (\mathsf{t}_{\text{gc}}.C_{\text{gc}} \parallel \mathsf{t}_1.\mathbf{T}(\mathbb{C}_1) \parallel \ldots \parallel \mathsf{t}_n.\mathbf{T}(\mathbb{C}_n), \mathsf{Id}, \mathsf{True})\\ &\qquad \preceq_{\alpha;\zeta\ltimes\zeta} (\mathsf{t}_{\text{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n, \mathsf{Id}, \mathsf{True}).\end{aligned}$$

其中 $\zeta \stackrel{\text{def}}{=} \zeta_{\text{gc}} \cap (\bigcap_{\mathsf{t}} \zeta(\mathsf{t}))$。

最后，由 RGSim 的适当性（推论 2.6），我们得到 $\mathsf{Correct}(\mathbf{T})$。 □

4.3　应用举例：Boehm 等人提出的并发垃圾收集算法

下面我们应用定理 4.1的验证框架证明 Boehm 等人提出的并发的标记 -清扫（mark-sweep）GC 算法 [23] 的正确性。该算法的变种已经在实际中使用（如 IBM JVM [38]）。

4.3.1　GC 算法概述

该 GC 主要分为标记阶段和清扫阶段，二者都是与用户程序并发执行的。任何时刻，堆中的活跃对象构成一个有向图，用户程序的指针变量所指向的对象构成根集（roots），堆中对象间的指针构成有向图的边。在标记阶段，GC 采用深度优先算法遍历并标记所有的可达对象。之后在清扫阶段，它会从堆的首地址开始，逐个对象地扫描整个堆，回收未被标记的对象。在 GC 遍历可达对象的过程中，对象之间的指向关系可能被用户线程改变，我们需要一个写拦截器来告诉 GC 这些改动。Boehm 等人的算法给每个对象一个额外的比特位——称作卡片（card），写拦截器会将正在修改的对象的卡片置位——称作弄脏（dirty）。然

```
 1  constant int WHITE, BLACK, BLUE; // colors
 2  constant int N; // total number of threads
 3  constant int M; // size of heap
 4
 5  Collection() {
 6    local mstk;
 7    while (true) {
 8      Initialize();
 9      Trace();
10      CleanCard();
11      atomic{ ScanRoot(); CleanCard(); }
12      Sweep();
13    }
14  }
15
16  Initialize() {
17    local i, c;
18    i := 1;
19    while (i <= M) {
20      i.dirty := 0;
21      c := i.color;
22      if (c = BLACK) { i.color := WHITE; }
23      i := i + 1;
24    }
25  }
26
27  Trace() {
28    local t, rt, i;
29    t := 1;
30    while (t <= N) {
31      rt := get_root(t);
32      foreach i in rt do { MarkAndPush(i); }
33      t := t + 1;
34      TraceStack();
35    }
36  }
37
38 TraceStack() {
39    local i, j;
40    while (!is_empty(mstk)) {
41      i := pop(mstk);
42      j := i.pt_1; MarkAndPush(j);
43      ...
44      j := i.pt_m; MarkAndPush(j);
45    }
46  }
```

图 4.1 Boehm 等人的 GC 代码（1）

```
47  MarkAndPush(i) {
48    local c;
49    if (i != 0) {
50      c := i.color;
51      if (c = WHITE) {
52        i.color := BLACK;
53        push(i, mstk);
54      }
55    }
56  }
57
58  CleanCard() {
59    local i, c, d;
60    i := 1;
61    while (i <= M) {
62      c := i.color;
63      d := i.dirty;
64      if (d = 1) {
65        i.dirty := 0;
66        if (c = BLACK) { push(i, mstk); }
67      }
68      i := i + 1;
69    }
70    TraceStack();
71  }
72
73  ScanRoot() {
74    local t, rt, i;
75    t := 1;
76    while (t <= N) {
77      rt := get_root(t);
78      foreach i in rt do { MarkAndPush(i); }
79      t := t + 1;
80    }
81  }
82
83  Sweep() {
84    local i, c;
85    i := 1;
86    while (i <= M) {
87      c := i.color;
88      if (c = WHITE) { free(i); }
89      i := i + 1;
90    }
91  }
```

图 4.2 Boehm 等人的 GC 代码（2）

```
update(x, fd, E) { // fd∈{pt1, ..., ptm}
  atomic{ x.fd := E; aux := x; }
  atomic{ x.dirty := 1; aux := 0; }
}
```

图 4.3 Boehm 等人的 GC 配套的写拦截器代码

后，在标记和清扫阶段之间，GC 会执行一个短暂的全局暂停（stop-the-world）阶段，在这个阶段中，GC 暂停所有的用户线程，然后从已标记的脏对象出发重新遍历并标记可达的对象。这一阶段称为卡片清理（card-cleaning）。这样，在清扫阶段之前，所有的可达对象都会被标记，从而保证了 GC 算法的正确性。

图 4.1和图 4.2展示了 GC 线程的代码。我们假设每个对象有 m 个指针域 $\mathrm{pt}_1, \ldots, \mathrm{pt}_m$，一个数据域 `data`，以及两个辅助域 `color` 和 `dirty`。其中颜色域 `color` 有三种可能的值：`BLACK`，`WHITE` 和 `BLUE`。前两种值指示 GC 是否已标记该对象：黑色（`BLACK`）对象已被标记，白色（`WHITE`）对象未被标记。尚未分配的对象是蓝色的（`BLUE`），它们既不会被标记也不会被回收，但之后可以被用户程序分配。新分配出的对象是黑色的；而回收对象就是将其置为蓝色。充当卡片的 `dirty` 域要么是 0（表示不脏）要么是 1（表示脏了）。此外，我们还假设用户线程的总数为 `N`（标识号为 1 至 `N`），且堆的地址空间为 [1..M]。

为了增强 GC 代码的可读性，在图 4.1和图 4.2中，我们将代码划分成多个函数（可以视作宏）。GC 线程调用图 4.1中的 `Collection()`，周期性地不断重复执行该函数内的循环体。在每个收集周期中，GC 首先调用 `Initialize()`，把堆中所有对象的卡片（`dirty` 域）清空，颜色（`color` 域）复位。初始化后，GC 调用 `Trace()`，进入标记阶段。图 4.1第 31 行的指令 `rt` := **get_root**(`t`) 读取用户线程 `t` 的局部存储区中的所有指针变量的值（即根集），存入集合 `rt`。我们假设它是原子执行的，实际 GC 的实现 [38] 往往会暂停一个用户线程来读取其根集。第 32 行的 **foreach** `i` **in** `rt` **do** C 对集合 `rt` 中的每个值 `i` 执行代码 C。之后的函数 `TraceStack()` 使用一个标记栈 `mstk` 来做深度优先遍历。为了简单起见，我们假设有原语指令 **push**(x, `mstk`) 和 x := **pop**(`mstk`) 直接操作 `mstk`。第 11 行的全局暂停阶段用 **atomic**{C} 实现，它原子地执行整个代码 C。这里需要用 `ScanRoot()` 重新读取根集：由于写拦截器并不作用在根集上，我们应当保守地假定根集可能被用户程序改变。最后在第 12 行的清扫阶段 `Sweep()`，GC 使用 **free**(x) 回收对象 x。该算法通常还有一个并发的卡片清理阶段，如图 4.1第 10 行调用 `CleanCard()`，它在第 11 行的全局暂停的卡片清理之前，可以缩短后者的暂停时间。

图 4.3展示了写拦截器的代码，它在修改对象的指针域后将其 `dirty` 域置

$$
\begin{array}{rl}
(\mathit{HExpr})\ \mathbb{E} & ::= x \mid n \mid \mathbf{nil} \mid \mathbb{E}{+}\mathbb{E} \mid \mathbb{E}{-}\mathbb{E} \mid \ldots \\
(\mathit{HBExp})\ \mathbb{B} & ::= \mathbf{true} \mid \mathbf{false} \mid \mathbb{E}{=}\mathbb{E} \mid !\mathbb{B} \mid \ldots \\
(\mathit{HInstr})\ \mathbb{c} & ::= \boldsymbol{print}(\mathbb{E}) \mid x{:=}\mathbb{E} \mid x := y.\mathrm{fd} \mid x.\mathrm{fd} := \mathbb{E} \mid x := \boldsymbol{new}() \\
(\mathit{HStmts})\ \mathbb{C} & ::= \boldsymbol{skip} \mid \mathbb{c} \mid \mathbb{C}_1;;\mathbb{C}_2 \mid \boldsymbol{if}\ \mathbb{B}\ \boldsymbol{then}\ \mathbb{C}_1\ \boldsymbol{else}\ \mathbb{C}_2 \mid \boldsymbol{while}\ \mathbb{B}\ \boldsymbol{do}\ \mathbb{C} \\
(\mathit{HProg})\ \mathbb{W} & ::= \mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \,|||\, \mathsf{t}_n.\mathbb{C}_n \\
(\mathit{HField})\ \mathrm{fd} & \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m, \mathtt{data}\} \qquad\qquad (\mathit{MutID})\ \mathsf{t} \in [1..N]
\end{array}
$$

(a) 编程语言

$$
\begin{array}{rlcl@{\qquad}rlcl}
(\mathit{Loc}) & l & \in & \{L_1, \ldots, L_M, \mathbf{nil}\} & (\mathit{HVal}) & \mathrm{V} & \in & \mathit{Int} \cup \mathit{Loc} \\
(\mathit{HStore}) & \mathbb{s} & \in & \mathit{PVar} \rightharpoonup \mathit{HVal} & (\mathit{HObj}) & \mathrm{O} & \in & \mathit{HField} \rightharpoonup \mathit{HVal} \\
(\mathit{HHeap}) & \mathbb{h} & \in & \mathit{Loc} \rightharpoonup \mathit{HObj} & (\mathit{HThrds}) & \Pi & \in & \mathit{MutID} \rightharpoonup \mathit{HStore} \\
(\mathit{HState}) & \Sigma & \in & \mathit{HThrds} \times \mathit{HHeap} & & & &
\end{array}
$$

(b) 程序状态

图 4.4 带有 GC 机制的高层语言和状态模型

位。这里我们为每个用户线程引入一个只写（write-only）的辅助变量 `aux`，记录该线程正在修改的对象。`aux` 仅仅是为了验证而引入的，证明结束我们就可以删除它而不会影响程序的正确性。它可帮助定义用户线程的保证条件，描述写拦截器的一些时序性质。例如，用户线程应保证在将对象 x 的某个指针域改为指向另一个对象 y 后，一定会先设置 x 的 `dirty` 域，才会去修改其他指针（特别是那些指向 y 的指针）。否则，GC 可能无法知道 y 是从 x 可达的而错误地回收 y。图 4.3中，当对象 x 的指针域被修改时，我们将 `aux` 置为 x；用户线程的 $\mathcal{G}$ 则保证当 $\mathtt{aux} = x$ 时只能设置 x 的 `dirty` 域（见图 4.12(b) 中的 $\mathcal{G}^{\mathsf{t}}_{\mathrm{set\_dirty}}$）。该 GC 算法并不使用读拦截器或分配拦截器。对象分配操作可以用一个普通的并发链表算法实现，但是为了更好地关注 GC 算法本身的验证，我们将对象分配操作抽象为一条原语指令 $x := \mathbf{new}()$，它可以原子地找到堆中的一个尚未分配的（即蓝色的）对象并分配给 x。

4.3.2　程序变换

我们首先在图 4.4与图 4.5中定义高层与低层的程序语言和机器模型。它们都是图 2.2中的通用语言的实例。

- 每个高层的抽象对象具有 m 个指针域和一个数据域，而每个低层的具体对象还具有两个辅助域 `color` 和 `dirty`。
- 图 4.6(a) 定义了高层的抽象 GC 线程的行为 AbsGCStep，它保证不会修改用户线程的局部存储区以及堆中的可达对象。这里我们用

$$
\begin{array}{rl}
(\mathit{LExpr})\ E & ::= x \mid n \mid E{+}E \mid E{-}E \mid \ldots \\
(\mathit{LBExp})\ B & ::= \textbf{true} \mid \textbf{false} \mid E{=}E \mid !B \mid \texttt{is\_empty}(x) \mid \ldots \\
(\mathit{LInstr})\ c & ::= \textbf{print}(E) \mid x{:=}E \mid x := y.\mathit{fd} \mid x.\mathit{fd} := E \mid x := \textbf{new}() \\
 & \quad \mid x := \textbf{get\_root}(y) \mid \textbf{free}(x) \mid \textbf{push}(x, y) \mid x := \textbf{pop}(y) \\
(\mathit{LStmts})\ C & ::= \textbf{skip} \mid c \mid C_1; C_2 \mid \textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2 \mid \textbf{while}\ (B)\ C \\
 & \quad \mid \textbf{atomic}\{C\} \mid \textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C \\
(\mathit{LProg})\ W & ::= \mathsf{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \parallel \mathsf{t}_1.C_1 \parallel \ldots \parallel \mathsf{t}_n.C_n \\
(\mathit{LField})\ \mathit{fd} & \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m, \texttt{data}, \texttt{color}, \texttt{dirty}\}
\end{array}
$$

(a) 编程语言

$$
\begin{array}{rcll}
(\mathit{LVal}) & v & \in & \mathit{Int} \cup \mathscr{P}(\mathit{Int}) \cup \mathit{Seq}(\mathit{Int}) \\
(\mathit{LObj}) & o & \in & \mathit{LField} \rightharpoonup \mathit{LVal} \\
(\mathit{LThrds}) & \pi & \in & (\mathit{MutID} \cup \{\mathsf{t}_{\mathrm{gc}}\}) \rightharpoonup \mathit{LStore}
\end{array}
\qquad
\begin{array}{rcll}
(\mathit{LStore}) & s & \in & \mathit{PVar} \rightharpoonup \mathit{LVal} \times \{0, 1\} \\
(\mathit{LHeap}) & h & \in & [1..M] \rightharpoonup \mathit{LObj} \\
(\mathit{LState}) & \sigma & \in & \mathit{LThrds} \times \mathit{LHeap}
\end{array}
$$

(b) 程序状态

图 4.5 实现 GC 算法的低层语言和状态模型

$\mathsf{Reachable}(l)(\Pi, \mathbb{h})$ 表示堆 $\mathbb{h}$ 中位于地址 l 的对象是从 Π 中的根集可达的。

- 低层的具体 GC 线程可以使用特权指令（如 $x := \textbf{get\_root}(y)$ 和 $\textbf{free}(x)$ 等）来控制用户线程，管理内存堆。

- 高层的用户线程使用指令 $x := y.\mathrm{fd}$ 读取对象的域，使用 $x.\mathrm{fd} := \mathbb{E}$ 将 $\mathbb{E}$ 的值写入对象的域，使用 $x := \textbf{\textit{new}}()$ 分配新对象。如果指令 $x.\mathrm{fd} := \mathbb{E}$ 修改的是指针域（即 $\mathrm{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}$），那么它在低层会被翻译为图 4.3中的写拦截器。注意这里 $\mathbb{E}$ 要么是空指针 **nil**，要么是某个指针变量。若高层的用户线程想将指针域 $y.\mathrm{fd}'$ 的值赋给指针域 $x.\mathrm{fd}$，则须先读取 $y.\mathrm{fd}'$ 的值到一个指针变量 z，再使用指令 $x.\mathrm{fd} := z$。

- 高层的编程语言是类型化的，堆地址与整数被视作不同类型的值。图 4.6(b) 展示了高层语言的操作语义，其中我们用 $\mathsf{sameType}(\mathrm{V}, \mathrm{V}')$ 表示 V 和 V′ 的值属于同一类型。

- 低层机器上，我们允许 GC 做指针算术运算，因此不区分地址和整数。低层的值 v 可以是整数，整数集合，或整数序列。我们用 $\mathscr{P}(\_)$ 表示幂集，用 $\mathit{Seq}(\_)$ 表示序列集。每个低层的变量都有一个额外的比特位，记录对应高层变量的类型信息（0 代表非指针，1 代表指针），以方便 GC 获得用户线程

$$
\begin{array}{lcl}
\mathsf{Root}(\mathsf{t}, S) & \stackrel{\text{def}}{=} & \lambda\Sigma.\ \Sigma = (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}_{\mathsf{t}}\}, \mathbb{h}) \wedge S = \{l \mid \exists x.\mathbb{s}_{\mathsf{t}}(x) = l\} \\
\mathsf{Edge}(l_1, l_2) & \stackrel{\text{def}}{=} & \lambda\Sigma.\ \Sigma = (\Pi, \mathbb{h}) \wedge \exists \mathrm{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathbb{h}(l_1)(\mathrm{fd}) = l_2 \\
\mathsf{Path}_k(l_1, l_2) & \stackrel{\text{def}}{=} & \begin{cases} l_1 = l_2 & \text{if } k = 0 \\ \exists l_3.\ \mathsf{Edge}(l_1, l_3) \wedge \mathsf{Path}_{k-1}(l_3, l_2) & \text{if } k > 0 \end{cases} \\
\mathsf{Path}(l_1, l_2) & \stackrel{\text{def}}{=} & \exists k.\ \mathsf{Path}_k(l_1, l_2) \\
\mathsf{Reachable}(\mathsf{t}, l) & \stackrel{\text{def}}{=} & \exists S, l'.\ \mathsf{Root}(\mathsf{t}, S) \wedge l' \in S \wedge \mathsf{Path}(l', l) \wedge l \neq \mathbf{nil} \\
\mathsf{Reachable}(l) & \stackrel{\text{def}}{=} & \exists \mathsf{t} \in [1..N].\ \mathsf{Reachable}(\mathsf{t}, l) \\
\mathsf{AbsGCStep} & \stackrel{\text{def}}{=} & \{((\Pi, \mathbb{h}), (\Pi, \mathbb{h}')) \mid \forall l.\ \mathsf{Reachable}(l)(\Pi, \mathbb{h}) \Longrightarrow \mathbb{h}(l) = \mathbb{h}'(l)\}
\end{array}
$$

(a) AbsGCStep 的定义

$$
\frac{\mathbb{s}(x) = l \quad \mathbb{h}(l) = \mathrm{O} \quad [\![\mathbb{E}]\!]_{\mathbb{s}} = \mathrm{V} \quad \mathrm{O}(\mathrm{fd}) = \mathrm{V}' \quad \mathsf{sameType}(\mathrm{V}, \mathrm{V}')}{(x.\mathrm{fd} := \mathbb{E}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_{\mathsf{t}} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h}\{l \rightsquigarrow \mathrm{O}\{\mathrm{fd} \rightsquigarrow \mathrm{V}\}\}))}
$$

$$
\frac{x \notin \mathit{dom}(\mathbb{s}) \quad \text{or} \quad \mathbb{s}(x) \notin \mathit{dom}(\mathbb{h}) \quad \text{or} \quad [\![\mathbb{E}]\!]_{\mathbb{s}} = \bot \quad \text{or} \quad \neg\mathsf{sameType}(\mathbb{h}(\mathbb{s}(x))(\mathrm{fd}), [\![\mathbb{E}]\!]_{\mathbb{s}})}{(x.\mathrm{fd} := \mathbb{E}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_{\mathsf{t}} \mathbf{abort}}
$$

$$
\frac{\begin{array}{c} l \notin \mathit{dom}(\mathbb{h}) \qquad l \neq \mathbf{nil} \qquad \mathbb{s}(x) = l' \qquad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow l\} \\ \mathbb{h}' = \mathbb{h} \uplus \{l \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow \mathbf{nil}, \ldots, \mathrm{pt}_m \rightsquigarrow \mathbf{nil}, \mathtt{data} \rightsquigarrow 0\}\} \end{array}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_{\mathsf{t}} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}'\}, \mathbb{h}'))}
$$

$$
\frac{\neg(\exists l.l \notin \mathit{dom}(\mathbb{h}) \wedge l \neq \mathbf{nil}) \quad \mathbb{s}(x) = l' \quad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow \mathbf{nil}\}}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_{\mathsf{t}} (\textbf{\textit{skip}}, (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}'\}, \mathbb{h}))}
$$

$$
\frac{x \notin \mathit{dom}(\mathbb{s}) \quad \text{or} \quad \neg\exists l.\mathbb{s}(x) = l}{(x := \textbf{\textit{new}}(), (\Pi \uplus \{\mathsf{t} \rightsquigarrow \mathbb{s}\}, \mathbb{h})) \longrightarrow_{\mathsf{t}} \mathbf{abort}}
$$

$$
\frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathsf{t}_i} (\mathbb{C}_i', \Sigma') \quad \text{or} \quad (\Sigma, \Sigma') \in \mathsf{AbsGCStep} \wedge \mathbb{C}_i' = \mathbb{C}_i}{(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow (\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i' \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma')}
$$

$$
\frac{(\mathbb{C}_i, \Sigma) \longrightarrow_{\mathsf{t}_i} \mathbf{abort}}{(\mathsf{t}_{\mathrm{gc}}.\mathit{AbsGC} \,|||\, \mathsf{t}_1.\mathbb{C}_1 \,|||\, \ldots \mathsf{t}_i.\mathbb{C}_i \ldots |||\, \mathsf{t}_n.\mathbb{C}_n, \Sigma) \longrightarrow \mathbf{abort}}
$$

(b) 部分操作语义规则

图 4.6 带有 GC 机制的高层机器

$$
\begin{array}{rcl}
[\![n]\!]_{(s,\mathit{tag})} & = & \begin{cases} n & \text{if } \mathit{tag} = 0 \text{ or } \mathit{tag} = 2 \\ 0 & \text{if } \mathit{tag} = 1 \text{ and } n = 0 \\ \bot & \text{otherwise} \end{cases} \\
[\![x]\!]_{(s,\mathit{tag})} & = & \begin{cases} n & \text{if } s(x) = (n, b) \text{ and } (\mathit{tag} = b \vee \mathit{tag} = 2) \\ \bot & \text{otherwise} \end{cases} \\
[\![E_1 + E_2]\!]_{(s,\mathit{tag})} & = & \begin{cases} n_1 + n_2 & \text{if } [\![E_1]\!]_{(s,\mathit{tag})} = n_1 \text{ and } [\![E_2]\!]_{(s,\mathit{tag})} = n_2 \\ & \quad \text{and } (\mathit{tag} = 0 \vee \mathit{tag} = 2) \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathtt{is\_empty}(x)]\!]_{(s,\mathit{tag})} & = & \begin{cases} \mathbf{true} & \text{if } \mathit{tag} = 0 \text{ and } s(x) = (\epsilon, 0) \\ \mathbf{false} & \text{if } \mathit{tag} = 0 \text{ and } s(x) = (n :: A, 0) \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

图 4.7 实现 GC 算法的低层机器上的表达式求值

$$\frac{\mathsf{t} \in [1..N] \quad s(x) = (\_, b) \quad [\![E]\!]_{(s,b)} = n \quad s' = s\{x \rightsquigarrow (n, b)\}}{(x := E, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (\_, b) \quad [\![E]\!]_{(s,2)} = n \quad s' = s\{x \rightsquigarrow (n, b)\}}{(x := E, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s'\}, h))}$$

$$\frac{\begin{array}{c} s(y) = (n_y, 1) \quad h(n_y)(\mathit{fd}) = n \quad s(x) = (\_, b) \\ \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \Longrightarrow b = 1 \quad \mathit{fd} \in \{\mathtt{data}\} \Longrightarrow b = 0 \quad s' = s\{x \rightsquigarrow (n, b)\} \end{array}}{(x := y.\mathit{fd}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h))}$$

$$\frac{\begin{array}{c} s(x) = (n, 1) \quad h(n) = o \quad \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \Longrightarrow [\![E]\!]_{(s,1)} = n' \\ \mathit{fd} \in \{\mathtt{data}\} \Longrightarrow [\![E]\!]_{(s,0)} = n' \quad \mathit{fd} \in \{\mathtt{color}, \mathtt{dirty}\} \Longrightarrow [\![E]\!]_{(s,2)} = n' \end{array}}{(x.\mathit{fd} := E, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h\{n \rightsquigarrow o\{\mathit{fd} \rightsquigarrow n'\}\}))}$$

$$\frac{s(y) = (\mathsf{t}, 0) \quad s(x) = (\_, 0) \quad \pi(\mathsf{t}) = s_{\mathsf{t}} \quad S = \{n \mid \exists x. s_{\mathsf{t}}(x) = (n, 1)\} \quad s' = s\{x \rightsquigarrow (S, 0)\}}{(x := \textbf{get\_root}(y), (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s'\}, h))}$$

$$\frac{x \in \mathit{dom}(s) \quad s(y) = (\emptyset, 0)}{(\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h))}$$

$$\frac{s(x) = (\_, b) \quad s(y) = (\{n_1, \ldots, n_k\}, 0) \quad s' = s\{x \rightsquigarrow (n_1, b)\}}{\begin{array}{l} (\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C,\ (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} \\ \qquad (C; y := y \backslash \{x\}; \textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C,\ (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s'\}, h)) \end{array}}$$

$$\frac{(C, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}}^{*} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h'))}{(\textbf{atomic}\{C\}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h'))}$$

$$\frac{\begin{array}{c} \mathsf{t} \in [1..N] \quad s(x) = (\_, 1) \quad h(n)(\mathtt{color}) = \mathtt{BLUE} \quad s' = s\{x \rightsquigarrow (n, 1)\} \\ h' = h\{n \rightsquigarrow \{\mathrm{pt}_1 \rightsquigarrow 0, \ldots, \mathrm{pt}_m \rightsquigarrow 0, \mathtt{data} \rightsquigarrow 0, \mathtt{color} \rightsquigarrow \mathtt{BLACK}, \mathtt{dirty} \rightsquigarrow 0\}\} \end{array}}{(x := \textbf{new}(), (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h'))}$$

$$\frac{\mathsf{t} \in [1..N] \quad s(x) = (\_, 1) \quad \neg(\exists n.\ h(n)(\mathtt{color}) = \mathtt{BLUE}) \quad s' = s\{x \rightsquigarrow (0, 1)\}}{(x := \textbf{new}(), (\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}} (\textbf{skip}, (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (n, 1) \quad h(n) = o}{(\textbf{free}(x), (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h\{n \rightsquigarrow o\{\mathtt{color} \rightsquigarrow \mathtt{BLUE}\}\}))}$$

$$\frac{s(x) = (n', b) \quad s(y) = (A, 0) \quad s' = s\{y \rightsquigarrow (n' :: A, 0)\}}{(\textbf{push}(x, y), (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s'\}, h))}$$

$$\frac{s(x) = (\_, b) \quad s(y) = (n :: A, 0) \quad s' = s\{x \rightsquigarrow (n, b), y \rightsquigarrow (A, 0)\}}{(x := \textbf{pop}(y), (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s\}, h)) \longrightarrow_{\mathsf{t}_{\mathsf{gc}}} (\textbf{skip}, (\pi \uplus \{\mathsf{t}_{\mathsf{gc}} \rightsquigarrow s'\}, h))}$$

图 4.8 实现 GC 算法的低层机器的部分操作语义规则

$$\mathbf{T}(\Sigma) \stackrel{\text{def}}{=} (\{\mathsf{t} \rightsquigarrow \mathbf{T}(\mathbb{s}) \mid (\mathsf{t} \rightsquigarrow \mathbb{s}) \in \Pi\} \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s_{\text{gc\_init}}\}, \mathbf{T}(\mathbb{h})), \quad \text{if } \Sigma = (\Pi, \mathbb{h}) \wedge \mathsf{WfState}(\Sigma)$$

$$\text{where } \mathsf{WfState}(\Pi, \mathbb{h}) \stackrel{\text{def}}{=} \forall l.\, \mathsf{Reachable}(l)(\Pi, \mathbb{h}) \Longrightarrow l \in \mathit{dom}(\mathbb{h})$$

$$s_{\text{gc\_init}} \stackrel{\text{def}}{=} \{\texttt{mstk} \rightsquigarrow \epsilon^{\text{np}}, \texttt{rt} \rightsquigarrow \emptyset^{\text{np}}, \texttt{i} \rightsquigarrow 0^{\text{p}}, \texttt{j} \rightsquigarrow 0^{\text{p}}, \texttt{c} \rightsquigarrow 0^{\text{np}}, \texttt{d} \rightsquigarrow 0^{\text{np}}, \texttt{t} \rightsquigarrow 0^{\text{np}}\}$$

$$\mathbf{T}(\mathbb{s})(x) \stackrel{\text{def}}{=} \begin{cases} n^{\text{np}} & \text{if } \mathbb{s}(x) = n \\ n^{\text{p}} & \text{if } \mathbb{s}(x) = l \wedge \mathsf{Loc2Int}(l) = n \\ 0^{\text{p}} & \text{if } x = \texttt{aux} \end{cases}$$

$$\mathbf{T}(\mathbb{h})(i) \stackrel{\text{def}}{=} \begin{cases} \{\text{pt}_1 \rightsquigarrow n_1, \ldots, \text{pt}_m \rightsquigarrow n_m, \texttt{data} \rightsquigarrow n, \texttt{color} \rightsquigarrow \texttt{WHITE}, \texttt{dirty} \rightsquigarrow 0\} \\ \qquad\qquad \text{if } \exists l.\, l \in \mathit{dom}(\mathbb{h}) \wedge \mathsf{Loc2Int}(l) = i \wedge 1 \le i \le M \\ \qquad\qquad \wedge\, \mathbb{h}(l) = \{\text{pt}_1 \rightsquigarrow l_1, \ldots, \text{pt}_m \rightsquigarrow l_m, \texttt{data} \rightsquigarrow n\} \\ \qquad\qquad \wedge\, \mathsf{Loc2Int}(l_1) = n_1 \wedge \ldots \wedge \mathsf{Loc2Int}(l_m) = n_m \\ \{\text{pt}_1 \rightsquigarrow 0, \ldots, \text{pt}_m \rightsquigarrow 0, \texttt{data} \rightsquigarrow 0, \texttt{color} \rightsquigarrow \texttt{BLUE}, \texttt{dirty} \rightsquigarrow 0\} \\ \qquad\qquad \text{if } \exists l.\, l \notin \mathit{dom}(\mathbb{h}) \wedge \mathsf{Loc2Int}(l) = i \wedge 1 \le i \le M \end{cases}$$

图 4.9 Boehm 等人的 GC 算法的初始状态变换

的根集。低层的用户线程仍然不可以做指针算术运算。图 4.7定义了表达式 E 在局部存储区 s 下的求值。此时我们还提供一个额外的记号 *tag* 指示 E 是否用做堆中对象地址：*tag* $= 1$ 表示用于地址，否则 *tag* $= 0$。当 *tag* $= 2$ 时，我们不关心 E 的使用场合。由于 GC 具有特权可混用整数和地址，对 GC 代码中用到的表达式，我们常常不关心它是否一定用做地址。图 4.8展示了部分低层操作语义规则。其中，为了形式化 **foreach** x **in** y **do** C 的语义，我们假设 x 和 y 都是不会被 C 修改的临时变量。在每轮迭代的开始，我们将 x 置为集合 y 中的任一元素；结束这轮 C 的执行后，我们就将这个元素从 y 中删去。当 y 变成空集时，**foreach** 循环终止。

- 我们假设堆的地址空间都是有限的，在高层仅有 M 个有效的地址，而在低层，堆的地址空间固定为 $[1..M]$。高层的用户程序用 **nil** 表示空指针，在低层机器上它被翻译为 0。我们假设高层地址与低层地址整数之间存在一个双射函数：

$$\mathsf{Loc2Int} : \mathit{Loc} \leftrightarrow [0..M]$$

  它满足 $\mathsf{Loc2Int}(\textbf{nil}) = 0$。

变换 **T** 如下定义。代码方面，高层的抽象 GC 线程变换为图 4.1 和图 4.2中的具体 GC 线程。用户程序的指令 $x.\text{fd} := \mathbb{E}$ 变换为写拦截器 $\texttt{update}(x, \text{fd}, \mathbf{T}(\mathbb{E}))$，其中 fd 是 x 的一个指针域。**T** 作用在表达式 $\mathbb{E}$ 上时保持其语法不变，除了当 $\mathbb{E}$ 是 **nil** 时返回 0。用户程序的其他指令和结构都保持不变。

$$
\begin{aligned}
\mathsf{store\_map}(s, \mathbb{s}) &\stackrel{\text{def}}{=} \forall x \neq \texttt{aux}.\ (\forall n.\ s(x) = n^{\text{np}} \iff \mathbb{s}(x) = n) \\
&\qquad \wedge (\forall n.\ s(x) = n^{\text{p}} \iff \exists l.\ \mathsf{Loc2Int}(l) = n \wedge \mathbb{s}(x) = l) \\
\mathsf{obj\_map}(o, \mathrm{O}) &\stackrel{\text{def}}{=} \exists n_1, \ldots, n_m, n, c, l_1, \ldots, l_m.\ \mathsf{Loc2Int}(l_1) = n_1 \wedge \ldots \wedge \mathsf{Loc2Int}(l_m) = n_m \\
&\qquad \wedge o = \{\text{pt}_1 \rightsquigarrow n_1, \ldots, \text{pt}_m \rightsquigarrow n_m, \texttt{data} \rightsquigarrow n, \texttt{color} \rightsquigarrow c, \texttt{dirty} \rightsquigarrow \_\} \\
&\qquad \wedge c \neq \texttt{BLUE} \wedge \mathrm{O} = \{\text{pt}_1 \rightsquigarrow l_1, \ldots, \text{pt}_m \rightsquigarrow l_m, \texttt{data} \rightsquigarrow n\}) \\
\mathsf{unalloc}(o, \mathbb{h}, l) &\stackrel{\text{def}}{=} (o = \{\text{pt}_1 \rightsquigarrow \_, \ldots, \text{pt}_m \rightsquigarrow \_, \texttt{data} \rightsquigarrow \_, \texttt{color} \rightsquigarrow \texttt{BLUE}, \texttt{dirty} \rightsquigarrow \_\}) \\
&\qquad \wedge l \notin \mathit{dom}(\mathbb{h}) \\
\mathsf{heap\_map}(h, \mathbb{h}) &\stackrel{\text{def}}{=} \forall i, l.\ 1 \leq i \leq M \wedge \mathsf{Loc2Int}(l) = i \\
&\qquad \implies \mathsf{obj\_map}(h(i), \mathbb{h}(l)) \vee \mathsf{unalloc}(h(i), \mathbb{h}, l) \\
\alpha &\stackrel{\text{def}}{=} \{((\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow \_\}, h), (\Pi, \mathbb{h})) \mid \\
&\qquad \forall \mathsf{t}.\ \mathsf{store\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t})) \wedge \mathsf{heap\_map}(h, \mathbb{h}) \wedge \mathsf{WfState}(\Pi, \mathbb{h})\}
\end{aligned}
$$

图 4.10 Boehm 等人的 GC 算法的 α 关系定义

我们还需要将高层的初始状态变换到低层。图 4.9定义了变换 $\mathbf{T}(\Sigma)$。

- 首先我们要求高层的初始状态是*良形的*（well-formed），即 $\mathsf{WfState}(\Sigma)$ 成立。它要求所有可达对象都不能是野指针（dangling pointer）。
- 高层地址通过双射函数 Loc2Int 翻译为整数，作为低层地址。
- 高层变量翻译到低层时用额外的比特位记录高层的类型信息（非指针用 0，指针用 1）。这里我们用 v^{np} 和 v^{p} 分别作为 $(v, 0)$ 和 $(v, 1)$ 的简写。
- 高层的对象翻译到低层时增加了 `color` 和 `dirty` 域，初值分别为 `WHITE` 和 0。我们用未分配的对象将低层地址空间 $[1..M]$ 中的其他地址填满，它们的 `color` 域为 `BLUE`，其他的域为 0。
- 低层的具体 GC 线程有一个初始的局部存储区 $s_{\text{gc\_init}}$，其中 GC 的整型和指针型变量都被初始化为 0，标记栈变量 `mstk` 初始化为 ϵ，根集变量 `rt` 初始化为 $\emptyset$。

为了证明 $\mathsf{Correct}(\mathbf{T})$，我们应用定理 4.1的验证框架，证明低层实现是对高层用户指令的精化，并用一个基于 Rely-Guarantee 的一元逻辑验证 GC 代码。

4.3.3　用户指令上的精化证明

我们首先定义 α 和 $\zeta(\mathsf{t})$ 关系。如图 4.10所示，α 要求低层与高层在局部存储区上的关系满足 store_map，在堆上的关系满足 heap_map。二者的定义反映了图 4.9中定义的状态变换。我们忽略高层不可见的结构的值，包括 GC 的局

部变量，已分配对象的 `color` and `dirty` 域，以及未分配对象的所有域。α 还要求高层状态是良形的。这里我们仍然用 Loc2Int 将地址和整数关联起来。

对于用户线程 t，在每个变换单元（即高层指令）前后都应成立的 $\zeta(\mathsf{t})$ 关系蕴涵 α 关系。它还要求辅助变量 `aux`（见图 4.3）的值为空指针。

$$\zeta(\mathsf{t}) \stackrel{\text{def}}{=} \alpha \cap \{((\pi, h), (\Pi, \mathbb{h})) \mid \pi(\mathsf{t})(\mathtt{aux}) = 0^{\mathrm{p}}\}.$$

为了方便定义用户指令的保证条件，我们首先引入一些分离逻辑断言来描述状态。如图 4.11所示，我们遵照 Parkinson 等人的工作 [39] 将程序变量视为资源。我们用 $\mathsf{own}_{\mathrm{p}}(x)$ 和 $\mathsf{own}_{\mathrm{np}}(x)$ 分别表示 x 为指针和非指针变量时当前线程对 x 的所有权（ownership）。断言在 (π, s, h) 下解释，其中 s 是当前线程的局部存储区，π 包含其他所有线程的局部存储区，h 是共享堆。我们用 $E_1.\mathit{fd} \mapsto E_2$ 描述仅含一个对象 E_1 的一个域 *fd* 的堆，这个域的值为 E_2。分离合取（separating conjunction）$p * q$ 表示 p 和 q 在不相交的状态上成立。图 4.11(c) 定义了两个状态的不相交并（disjoint union）。我们用 $f_1 \uplus f_2$ 表示定义域不相交的两个偏函数的并。由于堆 h 实际上是高阶函数，它先将地址映射到对象，对象再将域映射到值，所以我们可以用 uncurry 算子将堆转化为 uncurried 形式。我们用 $h_1 \oplus h_2$ 表示当 $\mathsf{uncurry}(h_1)$ 和 $\mathsf{uncurry}(h_2)$ 的定义域不相交时两个堆的并。在堆的不相交并，以及线程局部存储区的不相交并的基础上，我们定义了状态的不相交并。我们用 $E_1.\mathit{fd} \hookrightarrow E_2$ 作为 $(E_1.\mathit{fd} \mapsto E_2) * \mathbf{true}$ 的简写，用 $\circledast_{x \in S}.p(x)$ 表示集合 S 上的迭代分离合取（iterated separating conjunction）。尽管图 4.11中的断言记号是定义在低层状态上的，我们在高层也使用类似的记号，如 $\mathbb{E}_1.\mathrm{fd} \Mapsto \mathbb{E}_2$ 表示仅有一个对象的一个域的高层堆。

仿照 RGSep [15]，图 4.11(d) 定义了两种形式的动作（action）。$p \ltimes_{\mathsf{t}} q$ 表示从满足 p 的状态到满足 q 的状态的转换，其中只有当前线程 t 的局部存储区和共享堆可能被修改，其他线程的局部存储区是不变的。$p \ltimes_{\mathsf{t}} q$ provided p' 还保证不会修改状态中满足 p' 的部分。

图 4.12给出了高层用户指令和其低层实现的保证条件，它们的定义遵照代码的操作语义。我们用 $x^{\mathrm{p}} = n$ 作为 $(x = n) \wedge \mathsf{own}_{\mathrm{p}}(x)$ 的简写，用 $x^{\mathrm{np}} = n$ 作为 $(x = n) \wedge \mathsf{own}_{\mathrm{np}}(x)$ 的简写（在上下文清楚的情况下往往省略上标）。谓词 blueO 和 newO 分别表示蓝色的对象和新分配的对象，它们的定义在图 4.14中。这里每个动作只会访问用户线程的局部存储区和共享堆，而不会访问 GC 的局部存储区。

我们证明低层的写拦截器代码是对高层的指针更新指令的精化：

$$(\mathtt{update}(x, \mathrm{fd}, E), \mathcal{R}(\mathsf{t}), \mathcal{G}^{\mathsf{t}}_{\text{write\_barrier}}) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (x.\mathrm{fd} := \mathbb{E}, \mathbb{R}(\mathsf{t}), \mathbb{G}^{\mathsf{t}}_{\text{write\_pt}}),$$

$$
\begin{array}{rrcl}
(\mathit{PVarList}) & O & ::= & \bullet \mid x, O \\
(\mathit{StateAssert}) & p, q & \in & \mathit{LThrds} \times \mathit{LStore} \times \mathit{LHeap} \to \mathit{Prop}
\end{array}
$$

(a) 状态断言

$$
\begin{array}{lcl}
B & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ [\![B]\!]_{(s,2)} = \textbf{true} \\
\mathsf{emp}_h & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \mathit{dom}(h) = \emptyset \\
\mathsf{own}_{\text{np}}(x) & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \wedge s(x) = (\_, 0) \\
\mathsf{own}_{\text{p}}(x) & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \wedge s(x) = (\_, 1) \\
\mathsf{own}(x) & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \mathit{dom}(s) = \{x\} \\
p * q & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \exists \pi_1, s_1, h_1, \pi_2, s_2, h_2.\ p(\pi_1, s_1, h_1) \wedge q(\pi_2, s_2, h_2) \\
 & & \quad \wedge \pi = \pi_1 \oplus \pi_2 \wedge s = s_1 \uplus s_2 \wedge h = h_1 \oplus h_2 \\
\mathsf{t}.x = E & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \exists n, b.\ \pi(\mathsf{t})(x) = (n, b) \wedge [\![E]\!]_{(s,2)} = n \\
E_1.\mathit{fd} \mapsto E_2 & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \exists n, n'.\ [\![E_1]\!]_{(s,2)} = n' \wedge \mathit{dom}(h) = \{n'\} \\
 & & \quad \wedge h(n')(\mathit{fd}) = n \wedge \mathit{dom}(h(n')) = \{\mathit{fd}\} \wedge [\![E_2]\!]_{(s,2)} = n \\
E_1.\mathit{fd} \hookrightarrow E_2 & \stackrel{\text{def}}{=} & (E_1.\mathit{fd} \mapsto E_2) * \textbf{true} \\
O_{\text{np}}; O_{\text{p}} \Vdash p & \stackrel{\text{def}}{=} & (\mathsf{own}_{\text{np}}(x_1) * \ldots * \mathsf{own}_{\text{np}}(x_i) * \mathsf{own}_{\text{p}}(y_1) * \ldots * \mathsf{own}_{\text{p}}(y_j)) \wedge p \\
 & & \qquad \text{where } O_{\text{np}} = x_1, \ldots, x_i, \bullet \text{ and } O_{\text{p}} = y_1, \ldots, y_j, \bullet \\
x \in S & \stackrel{\text{def}}{=} & \exists X. S = X \uplus \{x\} \\
\circledast_{x \in S}.p(x) & \stackrel{\text{def}}{=} & (S = \phi \wedge \mathsf{emp}) \vee (\exists z, S'.\ (S = \{z\} \uplus S') \wedge (\circledast_{x \in S'}.p(x)) * p(z))
\end{array}
$$

(b) 状态断言的简化记号（其中 $\uplus$ 和 $\oplus$ 在下面定义）

$$
\begin{array}{lcl}
f_1 \perp f_2 & \stackrel{\text{def}}{=} & (\mathit{dom}(f_1) \cap \mathit{dom}(f_2) = \emptyset) \\
f_1 \uplus f_2 & \stackrel{\text{def}}{=} & \begin{cases} f_1 \cup f_2 & \text{if } f_1 \perp f_2 \\ \perp & \text{otherwise} \end{cases} \\
h_1 \oplus h_2 & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{curry}(\mathsf{uncurry}(h_1) \cup \mathsf{uncurry}(h_2)) & \text{if } \mathsf{uncurry}(h_1) \perp \mathsf{uncurry}(h_2) \\ \perp & \text{otherwise} \end{cases} \\
\pi_1 \oplus \pi_2 & \stackrel{\text{def}}{=} & \begin{cases} \{\mathsf{t} \rightsquigarrow (\pi_1(\mathsf{t}) \uplus \pi_2(\mathsf{t})) \mid \mathsf{t} \in \mathit{dom}(\pi_1)\} \\ \qquad \text{if } \mathit{dom}(\pi_1) = \mathit{dom}(\pi_2) \wedge\ \forall \mathsf{t} \in \mathit{dom}(\pi_1).\ \pi_1(\mathsf{t}) \perp \pi_2(\mathsf{t}) \\ \perp \qquad \text{otherwise} \end{cases} \\
\sigma_1 \oplus \sigma_2 & \stackrel{\text{def}}{=} & \begin{cases} (\pi, h) & \text{if } \sigma_1 = (\pi_1, h_1) \wedge \sigma_2 = (\pi_2, h_2) \wedge \pi_1 \oplus \pi_2 = \pi \wedge h_1 \oplus h_2 = h \\ \perp & \text{otherwise} \end{cases}
\end{array}
$$

(c) 不相交并

$$
\begin{array}{lcl}
p \ltimes_{\mathsf{t}} q & \stackrel{\text{def}}{=} & \{((\pi \uplus \{\mathsf{t} \rightsquigarrow s\}, h), (\pi \uplus \{\mathsf{t} \rightsquigarrow s'\}, h'))\ \mid \\
 & & \quad \exists s_1, h_1, s_2, h_2, s_1', h_1'.\ p(\pi, s_1, h_1) \wedge q(\pi, s_1', h_1') \\
 & & \quad \wedge (s = s_1 \uplus s_2) \wedge (h = h_1 \uplus h_2) \wedge (s' = s_1' \uplus s_2) \wedge (h' = h_1' \uplus h_2)\} \\
p \ltimes_{\mathsf{t}} q \text{ provided } p' & \stackrel{\text{def}}{=} & (p \ltimes_{\mathsf{t}} q) \cap ((p * p') \ltimes_{\mathsf{t}} (q * p'))
\end{array}
$$

(d) 动作

图 4.11 基本断言的定义

$$
\begin{array}{lcl}
\mathbb{G}^{\mathsf{t}}_{\text{assgn\_int}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = n' \wedge \mathsf{emp}_h) \\
\mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}} & \stackrel{\text{def}}{=} & \exists x, l, l'.\ (x = l \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = l' \wedge \mathsf{emp}_h) \\
 & & \quad \mathsf{provided}\ (l' = \mathbf{nil} \vee \exists y.\ y = l' \vee \exists y, \mathrm{fd}.\ y.\mathrm{fd} \Mapsto l') \\
\mathbb{G}^{\mathsf{t}}_{\text{write\_data}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x.\mathtt{data} \Mapsto n) \ltimes_{\mathsf{t}} (x.\mathtt{data} \Mapsto n') \\
\mathbb{G}^{\mathsf{t}}_{\text{write\_pt}} & \stackrel{\text{def}}{=} & \exists x, \mathrm{fd}, l, l'.\ (x.\mathrm{fd} \Mapsto l) \ltimes_{\mathsf{t}} (x.\mathrm{fd} \Mapsto l')\ \ \mathsf{provided}\ \ (l' = \mathbf{nil} \vee \exists y.\ y = l') \\
\mathbb{G}^{\mathsf{t}}_{\text{new}} & \stackrel{\text{def}}{=} & \exists x.\ (x = \_ \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x = l \wedge l.\mathrm{pt}_1 \Mapsto \mathbf{nil} * \ldots * l.\mathrm{pt}_m \Mapsto \mathbf{nil} * l.\mathtt{data} \Mapsto 0) \\
\\
\mathbb{G}(\mathsf{t}) & \stackrel{\text{def}}{=} & \mathbb{G}^{\mathsf{t}}_{\text{assgn\_int}} \cup \mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\text{write\_data}} \cup \mathbb{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathbb{G}^{\mathsf{t}}_{\text{new}}
\end{array}
$$

(a) 高层保证条件

$$
\begin{array}{lcl}
\mathcal{G}^{\mathsf{t}}_{\text{assgn\_int}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{np}} = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x^{\mathrm{np}} = n' \wedge \mathsf{emp}_h)\ \ \mathsf{provided}\ \ (\mathtt{aux}^{\mathrm{p}} = 0) \\
\mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{p}} = n \wedge \mathsf{emp}_h) \ltimes_{\mathsf{t}} (x^{\mathrm{p}} = n' \wedge \mathsf{emp}_h) \\
 & & \quad \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0 * (n' = 0 \vee \exists y.\ y^{\mathrm{p}} = n' \\
 & & \qquad\qquad \vee \exists y, \mathit{fd}.\ \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \wedge y.\mathit{fd} \mapsto n' \vee n = n')) \\
\mathcal{G}^{\mathsf{t}}_{\text{write\_data}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x.\mathtt{data} \mapsto n) \ltimes_{\mathsf{t}} (x.\mathtt{data} \mapsto n')\ \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0) \\
\mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} & \stackrel{\text{def}}{=} & \exists x, \mathit{fd}, n, n'.\ (\mathtt{aux}^{\mathrm{p}} = 0 * x.\mathit{fd} \mapsto n) \ltimes_{\mathsf{t}} (\mathtt{aux}^{\mathrm{p}} = x * x.\mathit{fd} \mapsto n') \\
 & & \quad \mathsf{provided}\ ((n' = 0 \vee \exists y.\ y^{\mathrm{p}} = n') \wedge \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}) \\
\mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}} & \stackrel{\text{def}}{=} & \exists n.\ (\mathtt{aux}^{\mathrm{p}} = n * n.\mathtt{dirty} \mapsto \_) \ltimes_{\mathsf{t}} (\mathtt{aux}^{\mathrm{p}} = 0 * n.\mathtt{dirty} \mapsto 1) \\
\mathcal{G}^{\mathsf{t}}_{\text{new}} & \stackrel{\text{def}}{=} & \exists x, n, n'.\ (x^{\mathrm{p}} = n * \mathsf{blueO}(n')) \ltimes_{\mathsf{t}} (x^{\mathrm{p}} = n' * \mathsf{newO}(n'))\ \mathsf{provided}\ (\mathtt{aux}^{\mathrm{p}} = 0) \\
\\
\mathcal{G}(\mathsf{t}) & \stackrel{\text{def}}{=} & \mathcal{G}^{\mathsf{t}}_{\text{assgn\_int}} \cup \mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{write\_data}} \cup \mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}} \cup \mathcal{G}^{\mathsf{t}}_{\text{new}}
\end{array}
$$

(b) 低层保证条件

图 4.12 用户指令的保证条件定义

其中 $\mathcal{G}^{\mathsf{t}}_{\text{write\_barrier}} \stackrel{\text{def}}{=} \mathcal{G}^{\mathsf{t}}_{\text{write\_pt}} \cup \mathcal{G}^{\mathsf{t}}_{\text{set\_dirty}}$，它是两步实现的写拦截器的保证条件。$\mathbb{G}^{\mathsf{t}}_{\text{write\_pt}}$ 则是原子的高层写操作的保证条件。$\mathcal{R}(\mathsf{t})$ 和 $\mathbb{R}(\mathsf{t})$ 已在第 4.2节中的 (4.2) 式定义。对于其他高层指令，其低层实现保持了指令的语法不变，因此对应的精化证明较为简单。例如，我们可以证明：

$$
(x := \mathbf{new}(), \mathcal{R}(\mathsf{t}), \mathcal{G}^{\mathsf{t}}_{\text{new}} \cup \mathcal{G}^{\mathsf{t}}_{\text{assgn\_pt}}) \preceq_{\alpha;\zeta(\mathsf{t})\ltimes\zeta(\mathsf{t})} (x := \mathbf{new}(), \mathbb{R}(\mathsf{t}), \mathbb{G}^{\mathsf{t}}_{\text{new}} \cup \mathbb{G}^{\mathsf{t}}_{\text{assgn\_pt}}).
$$

4.3.4　GC 代码的验证

我们用一个一元逻辑来验证 GC 线程，这里证明的细节与精化验证技术无关，所以下面我们仅快速地浏览断言语言的定义、一元逻辑的设计、GC 代码的规范、关键程序点的断言以及证明的主要结构。注意，尽管这里不涉及精化验证，但正是 RGSim 使我们能够得到定理 4.1，从而将一元逻辑下的 GC 验证与精化证明联系起来。

$$\frac{}{\{(x^{\text{np}} = X') * (1 \le y^{\text{np}} \le N)\} x := \textbf{get\_root}(y) \{(x^{\text{np}} = X) * (1 \le y^{\text{np}} \le N \wedge \mathsf{root}(y, X))\}}$$

$$\frac{}{\{x.\texttt{color} \mapsto \_\} \textbf{free}(x) \{x.\texttt{color} \mapsto \texttt{BLUE}\}}$$

$$\frac{}{\{y, O_{\text{np}}; O_{\text{p}} \Vdash x = X \wedge y = Y\} \textbf{push}(x, y) \{y, O_{\text{np}}; O_{\text{p}} \Vdash x = X \wedge y = X :: Y\}}$$

$$\frac{}{\{y, O_{\text{np}}; O_{\text{p}} \Vdash x = X \wedge y = X' :: Y\} x := \textbf{pop}(y) \{y, O_{\text{np}}; O_{\text{p}} \Vdash x = X' \wedge y = Y\}}$$

$$\frac{\{p\}C\{q\} \quad (p \ltimes q) \Rightarrow \mathcal{G}}{\mathsf{Id}; \mathcal{G} \vdash \{p\}\textbf{atomic}\{C\}\{q\}} \qquad \frac{\mathsf{Id}; \mathcal{G} \vdash \{p\}\textbf{atomic}\{C\}\{q\} \quad \mathsf{Sta}(\{p, q\}, \mathcal{R})}{\mathcal{R}; \mathcal{G} \vdash \{p\}\textbf{atomic}\{C\}\{q\}}$$

$$\frac{p \Rightarrow \mathsf{own}_{\text{np}}(y) * \textbf{true} \quad \mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x) \wedge x \in y\}\, C; y := y \backslash \{x\}\, \{p * \mathsf{own}(x)\}}{\mathcal{R}; \mathcal{G} \vdash \{p * \mathsf{own}(x)\}\textbf{foreach}\ x\ \textbf{in}\ y\ \textbf{do}\ C\{p * \mathsf{own}(x) \wedge y = \emptyset\}}$$

图 4.13 用于 GC 代码验证的逻辑规则

用于验证 GC 线程的一元程序逻辑是在标准的 Rely-Guarantee 逻辑 [22] 的基础上，针对图 4.5的 GC 实现语言做了少量改动。我们已经在图 4.11中定义了基本的断言语言。图 4.13展示了部分推理规则。图的上半部分是用于顺序语句的规则。绝大部分与分离逻辑 [31] 的规则 一样，所以在此省略。图中展示的是我们为 GC 特权指令（如 $x := \textbf{get\_root}(y)$）增加的规则，以及专门针对我们的低层机器模型设计的规则（如 $\textbf{free}(x)$ 仅仅将对象 x 的 `color` 置为 `BLUE`）。图的下半部分是并发规则，遵循标准的 Rely-Guarantee 推理。容易证明该逻辑对于图 4.8的低层操作语义的可靠性，此处我们省略具体的证明过程。

下面我们定义 GC 的前条件 p_{gc} 和保证条件 $\mathcal{G}_{\text{gc}}$。GC 应从良形的低层状态开始执行，即 $p_{\text{gc}} \stackrel{\text{def}}{=} \mathsf{wfstate}$。对应于高层的 WfState 定义（见图 4.9），低层的 wfstate 要求所有可达对象都不能是蓝色的，即

$$\mathsf{wfstate} \stackrel{\text{def}}{=} \circledast_{x \in [1..M]}.\mathsf{obj}(x) \wedge (\forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{notBlue}(x)),$$

其中 $\mathsf{obj}(x)$，$\mathsf{reachable}(x)$ 和 $\mathsf{notBlue}(x)$ 都在图 4.14中定义。$\mathsf{obj}(x)$ 表示 x 是一个低层堆地址，具有 $\text{pt}_1, \ldots, \text{pt}_m$，`data`，`color` 和 `dirty` 域；$\mathsf{reachable}(x)$ 的定义类似高层的 $\mathsf{Reachable}(l)$（见图 4.6）。容易看出，从高层初始状态经过变换 **T** 得到的低层初始状态（见图 4.9）都是良形的。$\mathcal{G}_{\text{gc}}$ 的定义如下：

$$\begin{aligned}\mathcal{G}_{\text{gc}} \stackrel{\text{def}}{=} \{&((\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s\}, h), (\pi \uplus \{\mathsf{t}_{\text{gc}} \rightsquigarrow s'\}, h')) \mid \\ &\forall n.\ \mathsf{reachable}(n)(\pi, h) \Longrightarrow \\ &\lfloor h(n) \rfloor = \lfloor h'(n) \rfloor \wedge h(n).\texttt{color} \neq \texttt{BLUE} \wedge h'(n).\texttt{color} \neq \texttt{BLUE}\}.\end{aligned}$$

$$
\begin{array}{lcl}
\mathsf{obj}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \_ * x.\mathtt{dirty} \mapsto \_ * \mathbf{true} \\
\mathsf{blueO}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto \_ * \ldots * x.\mathrm{pt}_m \mapsto \_ * x.\mathtt{data} \mapsto \_ * x.\mathtt{color} \mapsto \mathtt{BLUE} * x.\mathtt{dirty} \mapsto \_ \\
\mathsf{newO}(x) & \stackrel{\text{def}}{=} & x.\mathrm{pt}_1 \mapsto 0 * \ldots * x.\mathrm{pt}_m \mapsto 0 * x.\mathtt{data} \mapsto 0 * x.\mathtt{color} \mapsto \mathtt{BLACK} * x.\mathtt{dirty} \mapsto 0 \\
\mathsf{black}(x) & \stackrel{\text{def}}{=} & x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \\
\mathsf{white}(x) & \stackrel{\text{def}}{=} & x.\mathtt{color} \hookrightarrow \mathtt{WHITE} \\
\mathsf{dirty}(x) & \stackrel{\text{def}}{=} & x.\mathtt{dirty} \hookrightarrow 1 \\
\mathsf{notBlue}(x) & \stackrel{\text{def}}{=} & \exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{BLUE}) \\
\mathsf{notWhite}(x) & \stackrel{\text{def}}{=} & \exists c.\ (x.\mathtt{color} \hookrightarrow c \wedge c \neq \mathtt{WHITE}) \\
\mathsf{notDirty}(x) & \stackrel{\text{def}}{=} & x.\mathtt{dirty} \hookrightarrow 0 \\
\mathsf{root}(\mathsf{t}, S) & \stackrel{\text{def}}{=} & \lambda(\pi, s, h).\ \exists s_{\mathsf{t}}.\ s_{\mathsf{t}} = \pi(\mathsf{t}) \wedge S = \{n \mid \exists x.\ s_{\mathsf{t}}(x) = (n, 1) \wedge x \neq \mathtt{aux}\} \\
\mathsf{edge}(x, y) & \stackrel{\text{def}}{=} & \exists \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ (x.\mathit{fd} \hookrightarrow y) \\
\mathsf{path}_k(x, y) & \stackrel{\text{def}}{=} & \begin{cases} x = y & \text{if } k = 0 \\ \exists z.\ \mathsf{edge}(x, z) \wedge \mathsf{path}_{k-1}(z, y) & \text{if } k > 0 \end{cases} \\
\mathsf{path}(x, y) & \stackrel{\text{def}}{=} & \exists k.\ \mathsf{path}_k(x, y) \\
\mathsf{reachable}(\mathsf{t}, x) & \stackrel{\text{def}}{=} & \exists S, y.\ \mathsf{root}(\mathsf{t}, S) \wedge y \in S \wedge \mathsf{path}(y, x) \wedge x \neq 0 \\
\mathsf{reachable}(x) & \stackrel{\text{def}}{=} & \exists t \in [1..N].\ \mathsf{reachable}(t, x) \\
\mathsf{wfstate} & \stackrel{\text{def}}{=} & \circledast_{x \in [1..M]}.\mathsf{obj}(x) \wedge (\forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{notBlue}(x)) \\
\mathsf{whiteEdge}(x, \mathit{fd}, y) & \stackrel{\text{def}}{=} & (x.\mathit{fd} \hookrightarrow y) \wedge \mathsf{white}(y) \wedge \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\} \\
\mathsf{whiteEdge}(x, y) & \stackrel{\text{def}}{=} & \exists \mathit{fd}.\ \mathsf{whiteEdge}(x, \mathit{fd}, y) \\
\mathsf{todirty}(x, n) & \stackrel{\text{def}}{=} & \exists \mathsf{t}, S.\ (\mathsf{t}.\mathtt{aux} = x \wedge \mathsf{root}(\mathsf{t}, S) \wedge n \in S) \\
\mathsf{instk}(n, A) & \stackrel{\text{def}}{=} & \exists n', A'.\ A = n' :: A' \wedge (n = n' \vee \mathsf{instk}(n, A')) \\
\mathsf{stkBlack}(A) & \stackrel{\text{def}}{=} & \forall x.\ \mathsf{instk}(x, A) \Longrightarrow \mathsf{black}(x) \\
\mathsf{reachBlack} & \stackrel{\text{def}}{=} & \forall x.\ \mathsf{reachable}(x) \Longrightarrow \mathsf{black}(x) \\
\mathsf{ptfdSta}(x.\mathit{fd}, y) & \stackrel{\text{def}}{=} & \exists n.\ (x.\mathit{fd} \hookrightarrow n) \wedge (y = n \vee \mathsf{dirty}(x) \vee n = 0 \vee \mathsf{todirty}(x, n)) \\
\mathsf{newOSta}(x) & \stackrel{\text{def}}{=} & \mathsf{obj}(x) \wedge \mathsf{black}(x) \wedge \forall \mathit{fd} \in \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}.\ \mathsf{ptfdSta}(x.\mathit{fd}, 0) \\
\mathsf{rtBlack}(t) & \stackrel{\text{def}}{=} & \exists S.\ \mathsf{root}(t, S) \wedge \forall n \in S.\ \mathsf{black}(n) \\
\mathsf{rtBlack} & \stackrel{\text{def}}{=} & \forall t \in [1..N].\ \mathsf{rtBlack}(t) \\
\mathsf{markRt}(n) & \stackrel{\text{def}}{=} & \forall t \in [1..n].\ \mathsf{rtBlack}(t) \\
\mathsf{clearColor}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ (x.\mathtt{color} \hookrightarrow \mathtt{BLACK} \Longrightarrow \mathsf{newOSta}(x)) \\
\mathsf{clearDirty}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ \mathsf{notDirty}(x) \\
\mathsf{reclaim}(n) & \stackrel{\text{def}}{=} & \forall x \in [1..n].\ \mathsf{notWhite}(x) \\
\mathsf{reachInv} & \stackrel{\text{def}}{=} & \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, y) \\
 & & \quad \Longrightarrow \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \\
\mathsf{reachStk}(A) & \stackrel{\text{def}}{=} & \forall x, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, y) \\
 & & \quad \Longrightarrow \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \\
\mathsf{reachTomk}(A, x_p, S_f, x_n) & \stackrel{\text{def}}{=} & \\
 & & \forall x, \mathit{fd}, y.\ \mathsf{reachable}(x) \wedge \mathsf{black}(x) \wedge \mathsf{whiteEdge}(x, \mathit{fd}, y) \\
 & & \Longrightarrow \mathsf{dirty}(x) \vee \mathsf{todirty}(x, y) \vee \mathsf{instk}(x, A) \vee (x = x_p \wedge \mathit{fd} \in S_f) \vee (y = x_n)
\end{array}
$$

图 4.14 GC 代码验证中用到的断言

```
{wfstate}
Collection() {
  local mstk: Seq(Int);
  Loop Invariant:  {wfstate ∗ (own_np(mstk) ∧ mstk = ϵ)}
  while (true) {
    Initialize();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    Trace();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    CleanCard();
    {(wfstate ∧ reachInv) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    atomic{
      ScanRoot();
      { ∃X.(wfstate ∧ reachStk(X) ∧ stkBlack(X) ∧ rtBlack)
      { ∗ (own_np(mstk) ∧ mstk = X)
      CleanCard();
    }
    {(wfstate ∧ reachBlack) ∗ (own_np(mstk) ∧ mstk = ϵ)}
    Sweep();
  }
}
{false}
```

图 4.15 `Collection()` 的证明

GC 线程保证不会修改用户线程的局部存储区。对于可达对象，GC 也不会修改它们的指针域和数据域（这些域是高层用户线程可见的），或是回收它们。这里我们用 $\lfloor o \rfloor$ 将一个低层的对象 o 提升（lift）为一个高层的对象：

$$\lfloor o \rfloor \stackrel{\text{def}}{=} \{\mathrm{pt}_1 \rightsquigarrow o(\mathrm{pt}_1), \ldots, \mathrm{pt}_m \rightsquigarrow o(\mathrm{pt}_m), \mathtt{data} \rightsquigarrow o(\mathtt{data})\}\,.$$

我们可以证明 $\mathcal{G}_{\mathrm{gc}}$ 不比 AbsGCStep 具有更多的行为，即：

$$\mathcal{G}_{\mathrm{gc}} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ \mathsf{AbsGCStep}\,.$$

图 4.15展示了 GC 代码的主要证明结构。reachInv 是证明中用到的一个关键断言，它定义在图 4.14中。其含义是，若一个可达的黑色对象 x 指向一个白色对象 y，则要么 x 的卡片已经是脏的，要么存在某个用户线程即将把 x 弄脏（谓词 $\mathsf{todirty}(x, y)$ 成立）。后者这种情况出现在用户线程 t 刚完成写拦截器$\mathtt{update}(x, fd, y)$ 的第一步的时候，此时 $\mathsf{t}.\mathtt{aux} = x$，由用户线程的保证条件（图 4.12(b)）可知，线程 t 的下一步一定是弄脏对象 x 的卡片。

由于 GC 代码的每一条指令都是原子的，我们用图 4.13中针对 **atomic**$\{C\}$ 的规则来验证它们。注意 **atomic**$\{C\}$ 的第二条规则要求前后条件是稳定的（stable）。例如，读取对象指针域的语句的前后条件如下式 (4.5) 所示，其中后条件应考虑

到并发执行的用户线程可能修改该指针域。

$$\mathcal{R}_{\text{gc}};\mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \exists X, Y.\, (\texttt{j} = Y) * (\texttt{i.pt}_1 \hookrightarrow X) \right\} \\ \quad \texttt{j := i.pt1;} \\ \left\{ \exists X.\, (\texttt{j} = X) * \mathsf{ptfdSta}(\texttt{i.pt}_1, X) \right\} \end{array} \tag{4.5}$$

这里 $\mathsf{ptfdSta}(\texttt{i.pt}_1, X)$ 在图 4.14中定义。其含义为，要么 i 的 pt_1 域为 X，要么 i 已经或将要被弄脏。

类似地，由于用户线程分配新对象时会将对象的 color 域从 BLUE 更新为 BLACK，读取对象 color 域语句的稳定的后条件应考虑到这种修改，如下式 (4.6) 所示。其中 $\mathsf{newOSta}(\texttt{i})$ 在图 4.14中定义。它表示 i 指向一个新的对象，其 color 域为 BLACK，每个指针域要么是 0 要么已被更新（此时对象已经或将要被弄脏）。

$$\mathcal{R}_{\text{gc}};\mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \exists X, Y.\, (\texttt{c} = X) * (\texttt{i.color} \hookrightarrow Y) \right\} \\ \texttt{c := i.color;} \\ \left\{ \begin{array}{l} \exists X, Y.\, (\texttt{c} = X) * (\texttt{i.color} \hookrightarrow Y) \\ \wedge (X = Y \vee (X = \texttt{BLUE} \wedge \mathsf{newOSta}(\texttt{i}))) \end{array} \right\} \end{array} \tag{4.6}$$

模块 MarkAndPush(i) 在 GC 代码中被多次调用，因此下面我们首先给出它的普适的前后条件。如果对象 i 是白色的，MarkAndPush(i) 将它置为黑色，并压入标记栈中。

$$\mathcal{R}_{\text{gc}};\mathcal{G}_{\text{gc}} \vdash \begin{array}{l} \left\{ \begin{array}{l} \exists A.\, \mathsf{wfstate} \wedge \mathsf{reachTomk}(A, x_p, S_f, \texttt{i}) \\ \wedge\, \mathsf{stkBlack}(A) \wedge (\texttt{i} = 0 \vee \mathsf{obj}(\texttt{i})) \end{array} \right\} \\ \texttt{MarkAndPush(i);} \\ \left\{ \begin{array}{l} \exists A.\, \mathsf{wfstate} \wedge \mathsf{reachTomk}(A, x_p, S_f, 0) \\ \wedge\, \mathsf{stkBlack}(A) \wedge (\texttt{i} = 0 \vee \mathsf{notWhite}(\texttt{i})) \end{array} \right\} \end{array} \tag{4.7}$$

这里 $\mathsf{reachTomk}(A, x_p, S_f, x_n)$ 定义在图 4.14中。其含义为，如果一个可达的黑色对象 x 通过 *fd* 域指向一个白色对象 y，那么下列条件之一成立：

(1) $\texttt{dirty}(x) \vee \mathsf{todirty}(x)$：$x$ 已经或将要被弄脏。此条件与 reachInv 中的要求一致。

(2) $\mathsf{instk}(x, A)$：x 在栈 A 上。

(3) $x = x_p \wedge \mathit{fd} \in S_f$：$x$ 等于 x_p，且 *fd* 是 S_f 中的一个域。

```
{wfstate}
Initialize() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
  Loop Invariant: {(wfstate ∧ clearColor(i − 1) ∧ 1 ≤ i ≤ M + 1) ∗ own_np(c)}
  while (i <= M) { ... } // See Figure 4.1 for the full code
}
{wfstate ∧ reachInv}    // using Lemma 4.2
```

图 4.16 `Initialize()` 的证明

(4) $y = x_n$：y 等于 x_n。

条件 (2) 描述了遍历过程中对象 x 被涂黑且压入标记栈中的情况。我们定义 reachStk 来表达仅条件 (1) 或 (2) 成立的情况。条件 (3) 和 (4) 在下面用到它们的地方再具体讨论。

图 4.15中的每个收集周期开始时，状态是良形的（wfstate），且 GC 的局部存储区中的标记栈 `mstk` 是空的。GC 依次完成下列工作。

1. 并发环境下的初始化 `Initialize()`，见图 4.16。我们用 clearColor(n) 表示 GC 已经将堆中地址 1 至 n 的对象的颜色复位（即将黑色对象置为白色）。但此时仍可能有对象是黑色的，因为并发的用户线程可以进行对象分配，将蓝色对象置为黑色。我们可以证明当 GC 将堆中所有对象的颜色复位后，reachInv 成立，如下引理所示。

   引理 4.2. wfstate $\wedge$ clearColor$(M) \Longrightarrow$ reachInv.

   也就是说，初始化之后，如果可达的黑色对象 x 指向一个白色对象 y，则 x 一定是新分配的对象，其指针域已经被修改，且 x 已经或即将被弄脏。

2. 并发的标记阶段 `Trace()`，见图 4.17。

   (a) GC 首先调用 `MarkAndPush(i)` 标记根集对象并将它们入栈。我们需要下面两个引理将 (4.7) 式中 `MarkAndPush(i)` 的普适前后条件与调用该模块时的实际前后条件联系起来。这样我们可以复用 `MarkAndPush(i)` 的证明。

   引理 4.3. reachStk$(X) \Longrightarrow$ reachTomk$(X, 0, \emptyset, i)$.

   引理 4.4. reachTomk$(X, 0, \emptyset, 0) \Longrightarrow$ reachStk(X).

   (b) 然后 GC 调用 `TraceStack()` 进行深度优先遍历，见图 4.18。GC 每次从标记栈中弹出一个对象前，循环不变式 reachStk 成立。设标记栈的顶部对象 `i` 指向一个白色对象 x。GC 依次完成下列操作：

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

```
Trace() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```

Loop Invariant: $\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{t}) \wedge 1 \leq \mathtt{t} \leq N+1) * \mathsf{own}_{\mathsf{np}}(\mathtt{rt}) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) \end{array} \right\}$

```
  while (t <= N) {
    rt := get_root(t);
```

Foreach Invariant: $\{FInv\}$

```
    foreach i in rt do {
```

$\{FInv \wedge \mathtt{i} \in \mathtt{rt}\}$ ***// using Lemma 4.3***

```
      MarkAndPush(i);
```

$\{FInv \wedge \mathtt{i} \in \mathtt{rt}\}$ ***// using Lemma 4.4***

```
    }
    t := t + 1;
```

$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{t}) \wedge 1 \leq \mathtt{t} \leq N+1) * \mathsf{own}_{\mathsf{np}}(\mathtt{rt}) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) \end{array} \right\}$

```
    TraceStack();
```

$\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{t}) \wedge 1 \leq \mathtt{t} \leq N+1) * \mathsf{own}_{\mathsf{np}}(\mathtt{rt}) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) \end{array} \right\}$

```
  }
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

其中 $FInv \stackrel{\text{def}}{=}$

$$\begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{t}) \wedge 1 \leq \mathtt{t} \leq N) * (\mathsf{own}_{\mathsf{np}}(\mathtt{rt}) \wedge \forall n \in \mathtt{rt}.\ 0 \leq n \leq M) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) \end{array}$$

图 4.17 `Trace()` 的证明

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

```
TraceStack() {
  local i: [1..M], j: [0..M];
```

Loop Invariant: $\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) * \mathsf{own}_{\mathsf{p}}(\mathtt{j}) \end{array} \right\}$

```
  while (!is_empty(mstk)) {
    i := pop(mstk);
```

$\left\{ \begin{array}{l} \exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(\mathtt{i} :: X') \wedge \mathsf{stkBlack}(X') \wedge \mathsf{obj}(\mathtt{i})) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') * \mathsf{own}_{\mathsf{p}}(\mathtt{j}) \end{array} \right\}$

```
    j := i.pt1;
```

$\left\{ \begin{array}{l} \exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(\mathtt{i} :: X') \wedge \mathsf{stkBlack}(X') \wedge \mathsf{obj}(\mathtt{i}) \\ \qquad \wedge\ \mathsf{ptfdSta}(\mathtt{i.pt}_1, \mathtt{j}) \wedge (\mathtt{j} = 0 \vee \mathsf{obj}(\mathtt{j}))) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') \end{array} \right\}$

$\left\{ \begin{array}{l} \exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \{\mathtt{pt}_2, \ldots, \mathtt{pt}_m\}, \mathtt{j}) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge\ (\mathtt{j} = 0 \vee \mathsf{obj}(\mathtt{j})) \wedge 1 \leq \mathtt{i} \leq M) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') \end{array} \right\}$

// using Lemma 4.5

```
    MarkAndPush(j);
```

$\left\{ \begin{array}{l} \exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \{\mathtt{pt}_2, \ldots, \mathtt{pt}_m\}, 0) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge\ (\mathtt{j} = 0 \vee \mathsf{notWhite}(\mathtt{j})) \wedge 1 \leq \mathtt{i} \leq M) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') \end{array} \right\}$

```
    ...
    j := i.ptm; MarkAndPush(j);
```

$\left\{ \begin{array}{l} \exists X'.\ (\mathsf{wfstate} \wedge \mathsf{reachTomk}(X', \mathtt{i}, \emptyset, 0) \wedge \mathsf{stkBlack}(X') \\ \qquad \wedge\ (\mathtt{j} = 0 \vee \mathsf{notWhite}(\mathtt{j}))) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X') \end{array} \right\}$

```
  }
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

图 4.18 `TraceStack()` 的证明

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

```
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
```

Loop Invariant: $\{LInv\}$

```
  while (i <= M) { ... } // See Figure 4.1 for the full code
```

$\{LInv\}$

```
  TraceStack();
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) * \mathsf{own}_{\mathsf{np}}(\mathtt{c}) * \mathsf{own}_{\mathsf{np}}(\mathtt{d})\}$

```
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

其中 $LInv \stackrel{\text{def}}{=}$

$$\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X)) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)$$
$$*\ (\mathsf{own}_{\mathsf{p}}(\mathtt{i}) \wedge 1 \leq \mathtt{i} \leq M + 1) * \mathsf{own}_{\mathsf{np}}(\mathtt{c}) * \mathsf{own}_{\mathsf{np}}(\mathtt{d})$$

图 4.19 `CleanCard()` 的证明

i. 将 `i` 出栈。此时，指向 x 的黑色对象 `i` 就不在栈上了。

ii. 读取 `i` 的 pt_1 域到局部变量 `j` 中。此时 $\mathsf{ptfdSta}(\mathtt{i}.\mathrm{pt}_1, \mathtt{j})$ 成立（见 (4.5) 式）。此外，我们知道 x 要么是 `j`，要么被 `i` 通过 $\mathrm{pt}_2, \ldots, \mathrm{pt}_m$ 域所指向。因此，$\mathsf{reachTomk}(\mathtt{mstk}, \mathtt{i}, \{\mathrm{pt}_2, \ldots, \mathrm{pt}_m\}, \mathtt{j})$ 成立。形式化为下述引理：

引理 4.5.

A. $\mathsf{reachStk}(i :: X) \Longleftrightarrow \mathsf{reachTomk}(X, i, \{\mathrm{pt}_1, \ldots, \mathrm{pt}_m\}, 0)$;

B. $\mathsf{reachTomk}(X, i, S_f, 0) \Longrightarrow \mathsf{reachTomk}(X, i, S_f, j)$;

C. $\mathsf{reachTomk}(X, i, S_f, j) \wedge \mathsf{ptfdSta}(i.fd, j) \wedge fd \in S_f$
$\Longrightarrow \mathsf{reachTomk}(X, i, S_f \backslash \{fd\}, j)$.

iii. 调用 `MarkAndPush(j)`。我们可以再次复用该模块的证明。

iv. 标记 `i` 的其他孩子并将它们入栈。其证明与前面两步相似，我们不再详细讨论。最终，reachStk 成立。

我们保证了标记阶段后 reachInv 仍然成立。如果可达的黑色对象 x 指向某个白色对象，则 x 的指针域一定被用户线程修改了，x 已经或将要被弄脏。

3. 并发的卡片清理阶段 `CleanCard()`，见图 4.19。注意我们可以复用 `TraceStack()` 的证明。在该阶段结束时，reachInv 仍然成立。

4. 全局暂停的卡片清理阶段。

   (a) GC 首先调用 `ScanRoot()`，重新扫描根集并将根集对象入栈，见图 4.20。此时 reachStk 和 rtBlack 成立。其中 rtBlack 是指所有根集对象都是黑色的。此外，标记栈上的所有对象都一定是黑色的（记为

$\{(\mathsf{wfstate} \wedge \mathsf{reachInv}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

```
ScanRoot() {
  local t: [1..N], rt: Set(Int), i: [0..M];
  t := 1;
```

Loop Invariant: $\left\{ \begin{array}{l} \exists X.\ (Inv \wedge 1 \leq \mathtt{t} \leq N+1) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) * \mathsf{own}_{\mathsf{np}}(\mathtt{rt}) \end{array} \right\}$

```
  while (t <= N) {
    rt := get_root(t);
```

Foreach Invariant:

$\left\{ \begin{array}{l} \exists X, Y.\ (Inv \wedge 1 \leq \mathtt{t} \leq N \wedge \mathsf{root}(\mathtt{t}, Y) \wedge \forall n \in (Y \backslash \mathtt{rt}).\ \mathsf{black}(n) \wedge \mathtt{rt} \subseteq Y) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathsf{p}}(\mathtt{i}) \end{array} \right\}$

```
    foreach i in rt do { MarkAndPush(i); }
    t := t + 1;
  }
}
```

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

其中 $Inv \stackrel{\text{def}}{=} \mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{markRt}(\mathtt{t}-1)$

图 4.20 `ScanRoot()` 在原子块中的证明

$\{\exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X)\}$

```
CleanCard() {
  local i: [1..M], c: {BLACK, WHITE, BLUE}, d: {1, 0};
  i := 1;
```

Loop Invariant:

$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(\mathtt{i}-1) \\ \quad \wedge 1 \leq \mathtt{i} \leq M+1) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathsf{np}}(\mathtt{c}) * \mathsf{own}_{\mathsf{np}}(\mathtt{d}) \end{array} \right\}$

```
  while (i <= M) { ... } // See Figure 4.1 for the full code
```

$\left\{ \begin{array}{l} \exists X.\ (\mathsf{wfstate} \wedge \mathsf{reachStk}(X) \wedge \mathsf{stkBlack}(X) \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(M)) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = X) * \mathsf{own}_{\mathsf{np}}(\mathtt{c}) * \mathsf{own}_{\mathsf{np}}(\mathtt{d}) \end{array} \right\}$

```
  TraceStack();
```

$\left\{ \begin{array}{l} (\mathsf{wfstate} \wedge \mathsf{reachInv} \wedge \mathsf{rtBlack} \wedge \mathsf{clearDirty}(M)) \\ * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon) * \mathsf{own}_{\mathsf{np}}(\mathtt{c}) * \mathsf{own}_{\mathsf{np}}(\mathtt{d}) \end{array} \right\}$

```
}
```

$\{(\mathsf{wfstate} \wedge \mathsf{reachBlack}) * (\mathsf{own}_{\mathsf{np}}(\mathtt{mstk}) \wedge \mathtt{mstk} = \epsilon)\}$

图 4.21 `CleanCard()` 在原子块中的证明

$\{\mathsf{wfstate} \wedge \mathsf{reachBlack}\}$

```
Sweep() {
  local i: [1..M], c: {BLACK, WHITE, BLUE};
  i := 1;
```

Loop Invariant:

$\{(\mathsf{wfstate} \wedge \mathsf{reachBlack} \wedge \mathsf{reclaim}(\mathtt{i}-1) \wedge 1 \leq \mathtt{i} \leq M+1) * \mathsf{own}_{\mathsf{np}}(\mathtt{c})\}$

```
  while (i <= M) { ... } // See Figure 4.1 for the full code
}
```

$\{\mathsf{wfstate} \wedge \mathsf{reachBlack} \wedge \mathsf{reclaim}(M)\}$

图 4.22 `Sweep()` 的证明

stkBlack）。这里调用的 `MarkAndPush(i)` 是原子执行的，其前后条件与并发执行时的 (4.7) 式相同。

(b) 然后 GC 原子地执行 `CleanCard()` 清理卡片，见图 4.21。最终，标记栈变为空，所有可达的对象都被置为黑色（记为 reachBlack）。这里原子执行的 `TraceStack()` 的证明与并发执行时的证明类似，在此省略。

5. 并发的清扫阶段 `Sweep()`，见图 4.22。无论用户线程如何与 GC 交互执行，所有的白色对象都保持不可达。因此，GC 回收白色对象总是安全的，能够保证 $\mathcal{G}_{\mathrm{gc}}$。在清扫阶段结束时，状态仍然是良形的。

4.4 相关工作

Vechev 等人定义了由抽象 GC 生成并发 GC 代码的程序变换 [40]。之后，Pavlovic 等人提出了从抽象的规范生成具体的并发 GC 代码的精化 [41]。这些工作假定所有的用户程序操作都是原子的，然后关注如何在一个统一的框架中描述抽象 GC 或规范的各种实例。相比之下，我们给出的是一个通用的 GC 正确性定义和证明方法，既验证并发 GC 代码本身又验证它与用户程序之间的交互（注意拦截器的代码往往是细粒度的）。此外，他们的工作仅针对 GC，例如要求具体 GC 不能比抽象 GC 标记更多的对象等，因而并不适用于其他的精化应用。

Kapoor 等人用并发分离逻辑验证了 Dijkstra 的 GC 算法 [42]。为了说明 GC 的规范是有意义的，他们还验证了一个代表性的用户程序。而我们则将并发 GC 的验证问题归结为程序变换的验证，保证了所有的用户程序的行为都能被保持。

我们的并发 GC 验证框架（定理 4.1）受到了 McCreight 等人的工作 [43] 的启发，后者提出了一个验证全局暂停的 GC 和渐进式 GC 的框架，但并不支持并发 GC。

第 5 章　验证并发对象线性一致性的程序逻辑

本章我们验证并发对象实现的线性一致性（linearizability）[4]。第 1.1.2节中已经提到，最直观的验证方法需要我们找到对象实现的可线性化点（linearization point，简写为 LP）。然而，许多算法的 LP 无法静态确定，对它们的验证是线性一致性验证领域的经典难题。本章提出一种霍尔风格的程序逻辑，可以验证这些 LP 不固定的并发对象。我们还提出一种新的模拟关系，作为该程序逻辑的元理论（meta-theory）。新的模拟关系是对前两章讨论的 RGSim 关系的一种扩展，它具有可组合性，能够蕴涵一种上下文精化关系（contextual refinement），而后者与线性一致性等价。应用本章提出的程序逻辑，我们成功验证了 12 个经典算法，其中两个算法已经在 Java 并发包 `java.util.concurrent` 中使用。

我们首先在第 5.1节分析 LP 不固定的并发对象为验证带来的挑战，阐述我们的基本想法。然后，在第 5.2节介绍基本的技术设定，定义线性一致性和与它等价的上下文精化关系。第 5.3节和第 5.4节分别展示我们的程序逻辑和新的模拟关系。最后，我们在第 5.5节讨论几个有趣的例子，体现我们的程序逻辑的应用。

5.1　关键挑战与我们的解决方案

我们首先非形式地介绍一个简单的程序逻辑，它可以验证 LP 固定的并发对象的线性一致性。然后，我们对它进行扩展，以支持 LP 不固定的算法。我们还会讨论程序逻辑的元理论方面的问题，即如何保证程序逻辑是线性一致性的可靠验证手段。

5.1.1　验证 LP 固定的并发对象的程序逻辑

下面通过一个例子，介绍一个基于 LP 方法的简单而直观的程序逻辑。图 5.1(a) 展示了 Treiber 栈实现 [10] 的入栈操作（先忽略第 7′ 行的蓝色代码）。栈对象由 `S` 指向的链表实现，`push(v)` 操作使用`cas` （compare-and-swap）原语尝试更新 `S` 指针，直到成功。指令`cas(&S, t, x)` 比较 `S` 和 `t` 的值，若二者相等，则更新 `S` 的值为 `x`，返回 **true**；否则，什么都不做，直接返回 **false**。

验证线性一致性时，我们首先需要在代码中找到 LP。图 5.1(a) 中`push(v)` 方法的 LP 是在第 7 行的 `cas` 成功的时候。也就是说，成功的`cas` 对应于抽象

```
1  push(int v) {
2   local x, t, b;
3   x := new node(v);
4   do {
5     t := S;
6     x.next := t;
7     <b := cas(&S,t,x);
7'     if(b) linself;>
8   } while(!b);
9  }
```

(a) Treiber 栈

```
1  readPair(int i, j) {
2   local a, b, v, w;
3   while(true) {
4     <a := m[i].d; v := m[i].v;>
5     <b := m[j].d; w := m[j].v;
5'     trylinself;>
6     if(v = m[i].v) {
6'      commit(cid ↣ (end, (a,b)));
7       return (a, b); }
8   } }
9  write(int i, d) {
10   <m[i].d := d; m[i].v++;> }
```

(c) 一对数据的快照

```
1  push(int v) {
2   local p, him, q;
3   p := new threadDescriptor(cid, PUSH, v);
4   while(true) {
5     if (tryPush(v)) return;
6     loc[cid] := p;
7     him := rand(); q := loc[him];
8     if (q != null && q.id = him && q.op = POP)
9       if (cas(&loc[cid], p, null)) {
10        <b := cas(&loc[him], q, p);
10'        if(b) {lin(cid); lin(him);}>
11        if (b) return; }
12    ...
13  } }
```

(b) HSY 栈

图 5.1 可线性化点与插桩的辅助指令

的原子的 `PUSH(v)` 操作；而其他的任何程序步都不对应抽象操作。这里我们将抽象栈表示成一个值的序列 `Stk`，抽象的 `PUSH(v)` 就表示为`Stk := v::Stk`。那么，成功的`cas` 指令就是整个入栈操作起作用的唯一时刻，也就是`push(v)` 方法实现的可线性化点。

我们可以将上述推理过程嵌入一个传统的（一元）并发程序逻辑，如 Rely-Guarantee 逻辑 [22] 和并发分离逻辑 [44] 等。受 Vafeiadis 的工作的启发 [15]，我们将抽象操作 γ 和抽象状态 σ_a 作为具体状态 σ 的辅助状态，即程序状态变成 $(\sigma, (\gamma, \sigma_a))$。然后，我们在具体的对象实现代码的 LP 处插桩辅助代码 **linself**。从字面上看，**linself** 指“线性化自身”（linearize self）。它会执行辅助状态中的抽象操作 γ，更新抽象状态 σ_a，如下面的操作语义规则所示：

$$\frac{(\gamma, \sigma_a) \rightsquigarrow (\mathbf{end}, \sigma_a')}{(\mathbf{linself}, (\sigma, (\gamma, \sigma_a))) \longrightarrow (\mathbf{skip}, (\sigma, (\mathbf{end}, \sigma_a')))}$$

其中 $\rightsquigarrow$ 表示在抽象层对 γ 的执行，**end** 表示已执行完。在图 5.1(a) 的例子中，我

们将 **linself** 插入在第 7′ 行，它与 LP 的具体代码放在同一个原子块中，因此二者会同时执行。这里 $\langle C \rangle$ 表示原子块（即 C 是原子执行的）。我们可以扩展传统的（一元）并发程序逻辑，添加新的推理规则支持 **linself**，然后用它推理插桩后的代码。这样就完成了对线性一致性的验证。

上述想法虽然直观，却不能处理 LP 不固定的算法，包括使用帮助机制的算法，以及 LP 位置依赖未来执行的算法。下面我们借助两个具有代表性的算法，HSY 栈实现以及一对数据的快照算法，详细分析它们带来的挑战，并解释我们的解决方案。

5.1.2　帮助机制与 Pending 线程池

使用帮助机制的典型算法是 HSY 栈 [18]。图 5.1(b) 展示了其入栈操作实现的一部分。该算法的主要思想是让并发的入栈和出栈操作互相抵消。

图 5.1(b) 中的`push(v)` 方法开始时，线程会创建自己的*线程描述符*（第 3 行），它包含线程标识号、线程将要做的操作的名称、以及操作所需的参数。当前线程 `cid` 首先尝试 Treiber 栈的入栈操作（第 5 行），若成功则可立即返回。否则，它将自己的线程描述符写入一个全局共享的 `loc` 数组（第 6 行），准备和其他线程的操作互相抵消。抵消数组 `loc`$[1..n]$ 中，每个线程都有一个单元，放置指向该线程的描述符的指针。接着，当前线程随机地读取 `loc` 的某个单元 `him`（第 7 行），若 `him` 的线程描述符 `q` 告诉我们它正在做出栈操作，那么当前线程 `cid` 就可以尝试将自己和 `him` 互相抵消。抵消的尝试用到两个`cas` 指令。第一个`cas` 清除 `loc` 中 `cid` 的信息，使得不会有其他线程再与 `cid` 互相抵消（第 9 行）。第二个`cas` 试图将 `loc` 中 `him` 的单元标记为“已与 `cid` 抵消”（第 10 行）。若成功，则第二个`cas` 既是 `cid` 的入栈操作的 LP，也是 `him` 的出栈操作的 LP，且 `cid` 的入栈操作就发生在 `him` 的出栈操作之前。

从这个例子可以看出，帮助机制可能要求当前线程线性化其他线程的操作。但是第 5.1.1节介绍的简单逻辑不能表达这种情况。帮助机制也破坏了模块化性质，给单个线程的验证造成困难。在例子中，线程 `cid` 的具体的一步可以同时对应 `cid` 和 `him` 的抽象步，而对于线程 `him`，其环境的一步可能完成它的抽象操作。我们必须确保：关于 `him` 的抽象操作是否完成以及如何完成的方面，这两个线程 `cid` 和 `him` 总有一致的看法。例如，如果我们让 `cid` 的具体的一步线性化 `him` 的抽象出栈操作，那么 `him` 必须确实在做（且尚未完成）出栈操作。否则，`cid` 和 `him` 就无法一起并发执行。

我们扩展第 5.1.1节的简单逻辑以支持帮助机制。首先，我们引入一种新的辅助指令 **lin**(t) 来线性化一个特定的线程 t。例如在图 5.1(b) 中，我们在 LP 处

插入第 10′ 行的代码，在执行 LP 的具体代码的同时，依次执行 cid 的抽象入栈操作和 him 的抽象出栈操作。辅助状态也会相应扩展，既记录 cid 的抽象操作，也记录 him 的抽象操作。事实上，我们引入一个 pending 线程池 U 作为辅助状态，它是线程到其抽象操作的映射。U 描述了可能被其他线程帮忙的所有线程。**lin**(t) 的语义就定义在新的状态 $(\sigma, (U, \sigma_a))$ 下。类似前面讨论的 **linself** 的语义，**lin**(t) 执行 U 中线程 t 的抽象操作。

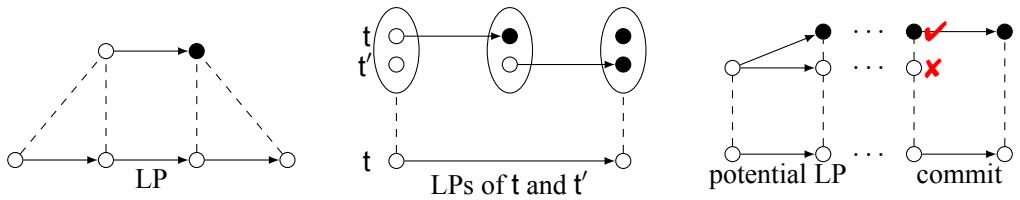
共享的 pending 线程池 U 让我们可以模块化地验证帮助机制。线程 cid 的具体的一步可以完成 U 中自身的操作以及 him 的操作，而并发执行的线程 him 也可以通过检查 U 来确定自己的操作是否已经被其他线程（如 cid）完成。也就是说，在单个线程的验证中，我们可以通过 U 获得其他线程的抽象操作的准确信息。注意 U 本身并不破坏模块化，因为 U 的信息实际上可以从程序的具体状态中推断出来。在 HSY 栈的例子中，我们可以由抵消数组中 him 的线程描述符得知它在做出栈操作。此时我们可以把 U 视作这个抵消数组的抽象表示。

5.1.3　依赖未来的 LP 与Try-Commit指令

另一大挑战是验证 LP 位置依赖未来执行的乐观算法。图 5.1(c) 给出了一个小例子，实现一对数据的快照（pair snapshot） [45]。该并发对象是一个数组 m。数组的每个单元包含两个域：数据域 d 和版本号 v。方法write(i,d)（见第 9 行）更新数组单元 i 的数据，同时将其版本号加 1。方法readPair(i,j) 想要原子地读取两个单元 i 和 j 的数据——即使存在并发的数据更新，仍要能对这一对数据做快照。具体来说，方法readPair(i,j) 首先分别在第 4 行和第 5 行读取单元 i 和 j 的数据，然后在第 6 行确认（validate）刚才读到的 i 的数据。如果 i 的版本号没有改变，则当线程在第 5 行读 j 的数据时，i 的数据并没有被更新。这说明第 4 行和第 5 行读到的是一致状态下的数据，因此线程可以返回。可以看出，该方法的 LP 在第 5 行读取 j 的数据的时候，但条件是第 6 行的确认成功。也就是说，我们是否在第 5 行线性化这一操作，依赖于未来第 6 行的不可预测的行为。

许多前人工作（如 [9, 15]）都提到，依赖未来的 LP 无法通过引入历史变量（history variable）来支持。历史变量是一种辅助变量，存储了过去执行中的值或事件的信息。但是此时我们需要的是不可预测的未来事件的信息。因此，人们提出了预言变量（prophecy variable） [9, 15]，它是历史变量的对偶，存储来自未来的信息。然而至今没有工作为预言变量给出适合霍尔风格的模块化推理的语义。

我们不采取预言变量的方法，而使用投机（speculation）的思想 [17]。对于具体实现中可能是 LP 的一步（如readPair 的第 5 行），我们投机地执行抽象操



(a) 简单模拟关系　　(b) Pending 线程池　　(c) 投机

图 5.2 验证线性一致性的模拟关系图

作，既记下执行后的结果，也保留原来的操作。之后，根据确认（如`readPair`的第 6 行）是否成功，我们留下合适的分支，丢掉错误的信息。

在程序逻辑中，我们引入两种新的辅助指令：用于投机的 **trylinself**，以及提交合适分支（满足断言 p 的分支）的 **commit**(p)。在图 5.1(c) 的例子中，我们插入第 5′ 行和第 6′ 行，其中 $\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b}))$ 的含义是，当前线程 `cid` 应该已经完成了它的抽象操作且会返回 $(\mathtt{a}, \mathtt{b})$。我们还扩展了辅助状态，记录投机后抽象操作和抽象状态的多种可能性。

此外，我们可以将投机的思想与 pending 线程池结合起来。不仅当前线程可以投机，pending 线程池中的抽象操作也可以投机执行。这样我们就可以验证一些极为巧妙的算法，这些算法中线程 t 的 LP 可能在其他线程 t′ 的代码中且依赖 t′ 的未来执行（如 RDCSS 算法 [46]）。我们会在第 5.5节验证这样一个例子。

5.1.4　作为元理论的模拟关系

我们可以将 LP 方法理解为建立了具体实现与抽象操作之间的弱模拟关系，如图 5.2(a) 所示。低层和高层的箭头分别表示具体实现与抽象操作的程序步，虚线表示模拟关系。高层的空心和实心节点分别表示抽象操作尚未做和已完成。低层有唯一的一步对应抽象的原子操作，它就是具体实现的 LP（图中标记“LP”）。这样的模拟关系形象地表达出了 LP 方法的思想，我们希望用它来证明程序逻辑的可靠性。特别地，我们想要一个模块化的模拟关系，可支持帮助机制和依赖未来的 LP，且蕴涵线性一致性。

首先，该模拟关系应具有可组合性：如果每个调用具体实现的线程都是对应抽象线程的模拟，那么整个多线程的具体程序也应是对应抽象程序的模拟。为此我们采用前面介绍的 RGSim 关系，将环境线程的行为以依赖/保证条件的形式作为模拟关系的参数，来获得可组合性。我们可以证明具有可组合性的 RGSim 关系蕴涵一种上下文精化关系，而后者与线性一致性等价。

为了支持帮助机制，模拟关系应当允许低层的一步对应环境线程的抽象操作。这需要我们知道环境线程的抽象代码和局部状态，但传统的单线程上的模拟关系（包括前面介绍的 RGSim）并不提供这些信息。为了解决这个问题，我们在模拟关系的抽象层引入 pending 线程池。图 5.2(b) 展示了新的模拟关系，其中线程 t 的具体的一步既线性化自己的操作，又线性化 pending 线程池中 t′ 的操作。这样的模拟关系直观地支持了帮助机制。

图 5.2(a) 和 (b) 中的模拟关系都是 forward 模拟关系，无法支持依赖未来的 LP。在 forward 模拟关系中，对于低层执行的每一步，我们都必须立刻决定它是否 LP，而不能将这一决定推迟到未来。许多前人工作（如 [9, 47–49]）都讨论了这个问题，也提到：为了支持依赖未来的 LP，我们必须引入 backward 模拟关系 或者 forward 和 backward 混合的模拟关系。这里我们采用投机的思想，设计 forward-backward 模拟关系 [48]。如图 5.2(c) 所示，低层的潜在 LP 对应于投机执行的抽象操作，较高的黑色节点表示已执行完抽象操作的结果，略低的白色节点仍然保留原来的抽象操作。之后的低层确认步对应于高层的提交步，我们留下正确的分支，丢弃错误的分支。

我们可以将上述想法结合起来，设计一种新的、具有可组合性的、支持不固定 LP 的模拟关系，作为我们的程序逻辑的元理论。我们会在第 5.4节形式化地定义该模拟关系。

5.2　基本设定

本节我们形式化地定义并发对象的线性一致性，并证明它与一种上下文精化关系等价。首先，我们定义支持并发对象的程序语言，它是第 2章讨论精化验证时介绍的通用语言（图 2.2）的实例。

5.2.1　支持并发对象的程序语言

如图 5.3所示，程序 W 包含多个并发执行的客户端（client）线程，每个线程都可以调用对象 Π 中声明的方法。方法的定义 (x, C) 包含形参 x 和方法体 C。为简单起见，我们假设 W 中仅有一个对象，且每个方法仅有一个参数。将我们的工作扩展到多个对象多个参数的情形并不困难。

方法结束时通过 **return** E 指令返回一个值给客户端程序。我们引入一个运行时（runtime）指令 **noret** 来中止（abort）那些执行结束却不返回的方法。该指令是自动添加在方法代码的末尾的，程序员不能直接使用它。**return** E 指令会先从 E 计算返回值 n，并归约为另一个运行时指令 **fret**(n)；然后 **fret**(n) 再将 n

$$
\begin{array}{rcll}
(\mathit{MName}) & f & \in & \mathit{String} \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E + E \mid \ldots \\
(\mathit{BExp}) & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid !B \mid \ldots \\
(\mathit{Instr}) & c & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \\
 & & \mid & x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{fret}(n) \mid \mathbf{noret} \\
 & & \mid & \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \parallel \ldots \parallel C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

图 5.3 支持并发对象的编程语言的语法

$$
\begin{array}{rcll}
(\mathit{ThrdID}) & \mathsf{t} & \in & \mathit{Nat} \\
(\mathit{Store}) & s & \in & \mathit{PVar} \rightharpoonup \mathit{Int} \qquad (\mathit{Heap})\ h \in \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma & ::= & (s, h) \\
(\mathit{CallStk}) & \kappa & ::= & (s_l, x, C) \mid \circ \\
(\mathit{ThrdPool}) & \mathcal{K} & ::= & \{\mathsf{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathsf{t}_n \rightsquigarrow \kappa_n\} \\
(\mathit{PState}) & \mathcal{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) \\
(\mathit{LState}) & \varsigma & ::= & (\sigma_c, \sigma_o, \kappa) \\
(\mathit{Evt}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & \mid & (\mathsf{t}, \mathbf{out}, n) \mid (\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T
\end{array}
$$

图 5.4 状态与事件

赋值给客户端程序接收返回值的变量，同时控制权返回客户端程序。把返回值的计算和控制权的返回作为两个分开的步骤，是为了允许环境在这两步之间的交互，模拟实际机器的执行过程。

其他的大多数指令是标准的（见 [24, 31]）。指令 $x := [E]$ 和 $[E] := E'$ 分别用来读取和更改内存单元。指令 $x := \mathbf{cons}(E_1, \ldots, E_n)$ 和 $\mathbf{dispose}(E)$ 则用来分配和释放内存。原子块 $\langle C \rangle$ 要求原子地执行 C。客户端程序可以用 $\mathbf{print}(E)$ 产生外部可观测事件。这里我们不允许对象的方法产生外部事件。为简单起见，我们还假设没有嵌套的方法调用。

图 5.4定义了程序状态和事件。全局状态 $\mathcal{S}$ 分为三个部分：客户端内存 σ_c，对象内存 σ_o，以及一个线程池 $\mathcal{K}$。内存 σ 包括存储区（store）和堆（heap），前者将变量映射到整数，后者将地址（自然数）映射到整数。除非调用对象方法，否则客户端程序仅能访问自己的内存 σ_c。线程池 $\mathcal{K}$ 将线程标识号 t 映射到其局部调用栈帧 κ。当线程不在执行方法时，其调用栈 κ 是空的（表示为 $\circ$）；否则，κ 是一个三元组 (s_l, x, C)，其中 s_l 存储了方法的形参和局部变量的值，x 是调用

者用来接收返回值的变量，C 是调用者的剩余代码（即方法返回后要执行的代码）。当所有线程的调用栈都是 $\circ$ 时，我们可以将线程池表示为 $\circledcirc$。我们还定义了线程对状态的局部视图 ς，以定义单线程的操作语义（见图 5.5(b)）。

图 5.4中定义的事件 e 既包含外部可观测事件，也包含外部不可观测的（可辅助定义线性一致性）。线程执行 $x := f(E)$ 时产生方法调用事件 (t, f, n)，其中 n 是实参 E 的值。执行方法体时产生 $(\mathsf{t}, \mathbf{obj})$。方法返回时产生返回事件 $(\mathsf{t}, \mathbf{ret}, n)$。$\mathbf{print}(E)$ 指令产生输出事件 $(\mathsf{t}, \mathbf{out}, n)$。其他的客户端指令产生 $(\mathsf{t}, \mathbf{clt})$。出错时，产生对象出错事件 $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ 或客户端出错事件 $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$。这里我们显式地区分对象实现的错误和客户端代码的错误，以方便讨论对象实现的安全性。只有输出事件以及这两种出错事件是外部可观测的。我们把方法调用事件、返回事件以及对象出错事件称作历史（history）事件，它们将用于定义线性一致性。我们用 $\mathsf{tid}(e)$ 得到事件 e 中的线程标识号。我们还定义了 e 上的若干谓词，说明事件的类型：

- e 是调用事件：$\mathsf{is\_inv}(e)$ 成立，当且仅当存在 t，f 和 n 使得 $e = (\mathsf{t}, f, n)$；
- e 是返回事件：$\mathsf{is\_ret}(e)$ 成立，当且仅当存在 t 和 n 使得 $e = (\mathsf{t}, \mathbf{ret}, n)$；
- e 是对象出错事件：$\mathsf{is\_obj\_abt}(e)$ 成立当且仅当存在 t 使得 $e = (\mathsf{t}, \mathbf{obj}, \mathbf{abort})$；
- e 是响应事件：$\mathsf{is\_res}(e)$ 成立，当且仅当 $\mathsf{is\_ret}(e)$ 或 $\mathsf{is\_obj\_abt}(e)$ 成立；
- e 是对象事件：$\mathsf{is\_obj}(e)$ 成立，当且仅当存在 t 使得 $e = (\mathsf{t}, \mathbf{obj})$，或 $\mathsf{is\_inv}(e)$，或 $\mathsf{is\_res}(e)$ 成立；
- e 是客户端出错事件：$\mathsf{is\_clt\_abt}(e)$ 成立，当且仅当存在 t 使得 $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$；
- e 是出错事件：$\mathsf{is\_abt}(e)$ 成立，当且仅当 $\mathsf{is\_obj\_abt}(e)$ 或 $\mathsf{is\_clt\_abt}(e)$ 成立；
- e 是客户端事件：$\mathsf{is\_clt}(e)$ 成立，当且仅当存在 t 和 n 使得 $e = (\mathsf{t}, \mathbf{clt})$，或 $e = (\mathsf{t}, \mathbf{out}, n)$，或 $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ 成立。

事件路径 T 是事件组成的有限序列。我们用 $T(i)$ 表示 T 的第 i 个事件，用 $|T|$ 表示 T 的长度。$T|_{\mathsf{t}}$ 表示 T 中所有线程标识号为 t 的事件构成的子路径。我们用 $\mathsf{get\_hist}(T)$ 表示 T 中所有历史事件构成的子路径；用 $\mathsf{get\_obsv}(T)$ 表示 T 中所有外部可观测事件构成的子路径。仅由历史事件构成的路径称为*历史*（history）。

$$\frac{}{(\textbf{let}\ \Pi\ \textbf{in skip} \parallel \ldots \parallel \textbf{skip}, \mathcal{S}) \longmapsto (\textbf{skip}, \mathcal{S})}$$

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (C_i', (\sigma_c', \sigma_o', \kappa')) \qquad \mathcal{K}' = \mathcal{K}\{i \rightsquigarrow \kappa'\}}{(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots C_i \ldots \parallel C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} (\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots C_i' \ldots \parallel C_n, (\sigma_c', \sigma_o', \mathcal{K}'))}$$

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} \textbf{abort}}{(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots C_i \ldots \parallel C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} \textbf{abort}}$$

(a) 程序语义

$$\frac{\Pi(f) = (y, C) \qquad [\![E]\!]_{s_c} = n \qquad x \in \mathit{dom}(s_c) \qquad \kappa = (\{y \rightsquigarrow n\}, x, \mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, f, n)}_{\mathsf{t},\Pi} (C; \textbf{noret}, ((s_c, h_c), \sigma_o, \kappa))}$$

$$\frac{f \notin \mathit{dom}(\Pi) \quad \text{or} \quad [\![E]\!]_{s_c}\ \mathit{undefined} \quad \text{or} \quad x \notin \mathit{dom}(s_c)}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{clt}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{\kappa = (s_l, x, C) \qquad s_c' = s_c\{x \rightsquigarrow n\}}{(\textbf{fret}(n), ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathsf{t}, \textbf{ret}, n)}_{\mathsf{t},\Pi} (C, ((s_c', h_c), \sigma_o, \circ))}$$

$$\frac{[\![E]\!]_{s_c} = n}{(\mathbf{E}[\,\textbf{print}(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{out}, n)}_{\mathsf{t},\Pi} (\mathbf{E}[\,\textbf{skip}\,], ((s_c, h_c), \sigma_o, \circ))}$$

$$\frac{(C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} (C', (s_o' \uplus s_l', h_o')) \qquad \mathit{dom}(s_l) = \mathit{dom}(s_l')}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C_c))) \xrightarrow{(\mathsf{t}, \textbf{obj})}_{\mathsf{t},\Pi} (C', (\sigma_c, (s_o', h_o'), (s_l', x, C_c)))}$$

$$\frac{\sigma_o = (s_o, h_o) \quad (C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} \textbf{abort}}{(C, (\sigma_c, \sigma_o, (s_l, x, C_c))) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}} \qquad \frac{(C, \sigma_c) \longrightarrow_{\mathsf{t}} (C', \sigma_c')}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathsf{t}, \textbf{clt})}_{\mathsf{t},\Pi} (C', (\sigma_c', \sigma_o, \circ))}$$

(b) 线程语义

$$\frac{[\![E]\!]_s = n}{(\mathbf{E}[\,\textbf{return}\ E\,], (s, h)) \longrightarrow_{\mathsf{t}} (\textbf{fret}(n), (s, h))} \qquad \frac{}{(\textbf{noret}, \sigma) \longrightarrow_{\mathsf{t}} \textbf{abort}}$$

$$\frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\textbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\textbf{fret}(n), \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} (\textbf{fret}(n), \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} \textbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathsf{t}} \textbf{abort}}$$

(c) 局部语义

图 5.5 支持并发对象的语言的操作语义规则

$$
\begin{aligned}
&\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \\
&\quad \{T \mid \exists W',\mathcal{S}'.\,(W,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} (W',\mathcal{S}') \vee (W,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} \textbf{abort}\} \\
&\mathcal{H}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\} \\
&\mathcal{O}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\}
\end{aligned}
$$

图 5.6 事件路径的生成

图 5.5给出了主要操作语义规则。我们定义了三种语义。$(W,\mathcal{S}) \xmapsto{e} (W',\mathcal{S}')$ 表示整个程序的全局语义；$(C,\varsigma) \xrightarrow{e}_{\mathsf{t},\Pi} (C',\varsigma')$ 表示使用对象 Π 的线程 t 的语义。我们还引入了一种局部语义 $(C,\sigma) \longrightarrow_{\mathsf{t}} (C',\sigma')$ 定义线程 t 在方法内/外的执行过程。当线程在方法内执行时，它仅访问对象内存和方法的局部变量及参数；当它在方法外执行时，它仅访问客户端内存。局部语义提升为线程语义时产生事件 $(\mathsf{t},\textbf{obj})$ 或 $(\mathsf{t},\textbf{clt})$。三种语义的执行都可能出错（**abort**）。为了定义后两种语义，我们还引入了执行上下文（execution context）**E**：

$$
(\textit{ExecContext})\quad \mathbf{E} \;::=\; [\,] \mid \mathbf{E};C
$$

这里 $[\,]$ 是将要执行的代码的位置。$\mathbf{E}[\,C\,]$ 表示把即将执行的 C 放置在上下文中。图 5.5中三种语义的定义都很直观。注意方法调用时我们将 **noret** 添加在方法体的末尾。**noret** 的执行总是 **abort**，因此安全的方法实现必须以 **return** 语句结尾。

图 5.6定义如何由程序执行生成事件路径。$\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]$ 表示由 W 从初始状态 $(\sigma_c,\sigma_o,\circledcirc)$ 开始执行产生的所有事件路径的前缀闭包，这里 $\circledcirc$ 指代一种特殊的线程池，它的所有线程的调用栈都是空的。图 5.6中用 $(W,\mathcal{S}) \xmapsto{T}{}^{*} (W',\mathcal{S}')$ 表示产生事件路径 T 的零或多步程序转换。我们还定义 $\mathcal{H}[\![W,(\sigma_c,\sigma_o)]\!]$ 和 $\mathcal{O}[\![W,(\sigma_c,\sigma_o)]\!]$ 分别得到执行产生的历史和可观测路径。

5.2.2　并发对象的规范与线性一致性

在定义线性一致性之前，我们先定义并发对象的抽象操作（称为对象的规范）。我们用 $\Pi_A \in \textit{ODecl}$ 表示对象的规范。我们要求 Π_A 中的每个方法都是原子的，且其执行永远是安全的和确定性的（deterministic）。为简单起见，我们用 γ 表示 Π_A 中方法的定义，即

$$
\begin{aligned}
\gamma \;&::=\; (x,\langle C\rangle) \\
\Pi_A \;&::=\; \{f_1 \rightsquigarrow \gamma_1,\ldots,f_n \rightsquigarrow \gamma_n\}
\end{aligned}
$$

注意我们允许原子块 $\langle C\rangle$ 内含有 **return** E 指令。在这种情况下，$\langle C\rangle$ 的执行会得到 **fret**(n)，如图 5.5(c) 定义的语义所示。不失一般性，我们假设抽象操作与具体实现用到的程序变量总是不同的。我们用 σ_a 表示抽象的对象内存。调用抽象

对象方法的客户端程序 W 的语义与上一节定义的（图 5.5）相同，我们只需将具体对象内存 σ_o 替换为抽象对象内存 σ_a。

线性一致性 [4] 的定义基于历史（history）的概念。历史是特殊的事件路径 T，它只含有历史事件，即方法调用事件和响应事件。

我们定义响应事件 e_2 匹配（match）调用事件 e_1，当且仅当它们的线程标识号相同：

$$\mathsf{match}(e_1, e_2) \stackrel{\text{def}}{=} \mathsf{is\_inv}(e_1) \wedge \mathsf{is\_res}(e_2) \wedge (\mathsf{tid}(e_1) = \mathsf{tid}(e_2))$$

历史 T 是*顺序的*（sequential），当且仅当 T 的第一个事件是调用事件，且（除了 T 的最后一个调用事件外）T 由相邻的、两两相匹配的调用事件和响应事件组成。它如下归纳定义：

$$\frac{}{\mathsf{seq}(\epsilon)} \qquad \frac{\mathsf{is\_inv}(e)}{\mathsf{seq}(e :: \epsilon)} \qquad \frac{\mathsf{match}(e_1, e_2) \quad \mathsf{seq}(T)}{\mathsf{seq}(e_1 :: e_2 :: T)}$$

如果每个线程上的子历史 $T|_{\mathsf{t}}$ 都是顺序的，则 T 是*良形的*（well-formed）：

$$\mathsf{well\_formed}(T) \stackrel{\text{def}}{=} \forall \mathsf{t}.\ \mathsf{seq}(T|_{\mathsf{t}})\,.$$

T 是*完整的*（complete），当且仅当它是良形的，且每个调用事件都有一个匹配的响应事件。如果调用事件之后没有相匹配的响应事件，则称该调用事件正在 pending。对于不完整的历史 T 中的 pending 的调用事件，我们按照标准的线性一致性定义 [4] 来处理：我们在 T 的末尾添加零个或多个返回事件，然后丢弃那些仍然 pending 的调用事件。这样我们就得到了一个由完整的历史构成的集合 $\mathsf{completions}(T)$。形式化地，我们需要先定义 $\mathsf{extensions}(T)$ 和 $\mathsf{truncate}(T)$：

定义 5.1 (历史的延展). $\mathsf{extensions}(T)$ *是在 T 的末尾添加返回事件得到的所有良形历史构成的集合，其归纳定义如下：*

$$\frac{\mathsf{well\_formed}(T)}{T \in \mathsf{extensions}(T)} \qquad \frac{T' \in \mathsf{extensions}(T) \quad \mathsf{is\_ret}(e) \quad \mathsf{well\_formed}(T' :: e)}{T' :: e \in \mathsf{extensions}(T)}$$

定义 5.2 (历史的完整化). $\mathsf{truncate}(T)$ *是 T 的最大的完整的子历史，其归纳定义如下：*

$$\mathsf{truncate}(\epsilon) \stackrel{\text{def}}{=} \epsilon$$

$$\mathsf{truncate}(e :: T) \stackrel{\text{def}}{=} \begin{cases} e :: \mathsf{truncate}(T) & \textit{if } \mathsf{is\_res}(e) \textit{ or } \exists i.\ \mathsf{match}(e, T(i)) \\ \mathsf{truncate}(T) & \textit{otherwise} \end{cases}$$

那么，$\mathsf{completions}(T) \stackrel{\text{def}}{=} \{\mathsf{truncate}(T') \mid T' \in \mathsf{extensions}(T)\}\,.$

现在我们可以定义两个良形历史之间的线性一致性关系，它是对象的线性一致性定义的核心。

定义 5.3 (历史之间的线性一致性关系). $T \preceq_{\mathsf{lin}} T'$ 当且仅当

1. $\forall \mathsf{t}.\, T|_{\mathsf{t}} = T'|_{\mathsf{t}}$;
2. 存在双射 $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ 使得 $\forall i.\, T(i) = T'(\pi(i))$ 且

$$\forall i, j.\, i < j \wedge \mathsf{is\_res}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j).$$

该定义说，如果 T' 是 T 的一种排列（permutation），且保持了单个线程上的事件的顺序（第 1 个条件）以及未并发的方法调用的顺序（第 2 个条件），那么 T 对于 T' 是线性一致的。

并发对象 Π 的线性一致性要求 Π 的每条历史都对于某条 合法的（legal）顺序历史是线性一致的。Π 的历史由所有可能的调用 Π 的客户端程序根据操作语义（图 5.5）执行生成。图 5.6 定义 $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ 表示程序 W 执行生成的历史。类似地，我们用 $\mathcal{H}[\![W, (\sigma_c, \sigma_a)]\!]$ 表示使用抽象对象时生成的历史。那么，合法的顺序历史 T 就是由某个客户端程序使用抽象对象 Π_A 时产生的顺序历史，它满足下述定义：

$$\begin{aligned}
&\Pi_A \rhd (\sigma_a, T) \stackrel{\text{def}}{=} \\
&\exists n, C_1, \ldots, C_n, \sigma_c.\, T \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a)]\!] \wedge \mathsf{seq}(T)
\end{aligned}$$

定义 5.4 (并发对象的线性一致性). 对象实现 Π 对于规范 Π_A 在精化映射（*refinement mapping*）φ 下是线性一致的，表示为 $\Pi \preceq_{\varphi} \Pi_A$，当且仅当

$$\begin{aligned}
&\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a, T. \\
&T \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \sigma_a) \\
&\implies \exists T_c, T'.\, T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\sigma_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'
\end{aligned}$$

其中 $\varphi : Mem \rightharpoonup Mem$ 将具体对象内存与抽象对象内存联系起来。

该定义的附加条件 $\varphi(\sigma_o) = \sigma_a$ 要求初始的 σ_o 应是某个有效的抽象对象内存 σ_a 的具体表示。例如，对于集合对象的实现，φ 可能要求 σ_o 为一个链表，代表 σ_a 中的抽象的数学集合。

5.2.3　与线性一致性等价的上下文精化关系

下面我们定义对象实现和规范之间的一种上下文精化关系，并证明它与线性一致性等价。该等价性是我们的程序逻辑能够保证线性一致性的基础。

简单来讲，上下文精化关系要求任何客户端程序在使用具体对象时产生的可观测行为都不会比使用抽象对象时产生的更多。我们用 $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$ 表示 W 执行过程中产生的可观测事件路径的集合（如图 5.6定义）。这里可观测事件包括输出事件、客户端出错和对象出错事件。

定义 5.5 (上下文精化关系). $\Pi \sqsubseteq_{\varphi} \Pi_A$ 当且仅当

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a.\ \varphi(\sigma_o) = \sigma_a \\ \Longrightarrow\ \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!] \subseteq \mathcal{O}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a)]\!]\,.\end{array}$$

定理 5.6 (线性一致性和上下文精化关系等价). $\Pi \preceq_{\varphi} \Pi_A \iff \Pi \sqsubseteq_{\varphi} \Pi_A$.

该定理的证明过程仿照 Filipović 等人的工作 [5] 以及 Gotsman 和 Yang 的工作 [50]。具体的证明可见附录 B。

该定理有两个方面的意义。一方面，由于线性一致性蕴涵上下文精化关系，在验证客户端程序的时候，我们就可以将它调用的线性一致的对象实现替换为相应的抽象操作，做到对客户端程序的模块化验证。另一方面，由于上下文精化关系也蕴涵线性一致性，我们就可以用验证上下文精化的技术（如 RGSim）来验证线性一致性。

5.3　验证线性一致性的程序逻辑

本节介绍我们设计的验证对象线性一致性的程序逻辑。我们首先在对象实现中插桩辅助代码，引入辅助状态，将具体实现与抽象对象联系起来。我们扩展一元的并发程序逻辑 LRG [24]，为辅助代码设计推理规则，定义描述辅助状态的断言。尽管我们的程序逻辑是基于 LRG 设计的，但我们的方法与基础逻辑的形式无关。我们也可以对其他的一元并发程序逻辑（如 RGSep [15]）做出类似的扩展，来验证对象实现的线性一致性。

注意本节提出的程序逻辑仅用于验证对象方法，它保证对象方法的具体实现是对应抽象原子操作的上下文精化，因此可以保证对象的线性一致性。有关整个程序（包括对象代码以及客户端代码）的验证，我们会在第 5.3.3节的结尾处讨论。

5.3.1　辅助代码和状态

图 5.7展示了插桩辅助代码后的程序的语法，以及相应的状态模型。正如我们在第 5.1节所描述的，新的程序状态 Σ 包含两个部分，除了实际的具体对象内存 σ 外，还包含辅助状态 Δ。非空集合 Δ 的每个元素是一个 (U, σ_a) 对，表示抽象层的一种投机情况。其中 σ_a 是该投机情况下的抽象对象内存。U 是 pending 线程池，记录各线程想要完成的操作。它将线程标识号映射到剩余的抽象操作 (γ, n) 或 $(\textbf{end}, n)$，前者是指抽象操作 γ 在参数 n 下执行，后者是指操作已经完成且返回值为 n。

$$
\begin{array}{rcll}
(\textit{InsStmt}) & \widetilde{C} & ::= & \textbf{skip} \mid c \mid \textbf{return}\ E \mid \textbf{noret} \mid \textbf{linself} \mid \textbf{lin}(E) \\
 & & \mid & \textbf{trylinself} \mid \textbf{trylin}(E) \mid \textbf{commit}(p) \mid \langle \widetilde{C} \rangle \\
 & & \mid & \widetilde{C}; \widetilde{C} \mid \textbf{if}\ (B)\ \widetilde{C}\ \textbf{else}\ \widetilde{C} \mid \textbf{while}\ (B)\{\widetilde{C}\} \\
(\textit{RelState}) & \Sigma & ::= & (\sigma, \Delta) \\
(\textit{SpecSet}) & \Delta & ::= & \{(U_1, \sigma_1), \ldots, (U_n, \sigma_n)\} \\
(\textit{PendThrds}) & U & ::= & \{\mathsf{t}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathsf{t}_n \rightsquigarrow \Upsilon_n\} \\
(\textit{AbsOp}) & \Upsilon & ::= & (\gamma, n) \mid (\textbf{end}, n) \\
(\textit{RelAss}) & p, q, I & ::= & B \mid \textit{own}(x) \mid \boldsymbol{own}(x) \mid \mathsf{emp} \\
 & & \mid & E \mapsto E \mid E \Mapsto E \mid E \rightarrowtail (\gamma, E) \mid E \rightarrowtail (\textbf{end}, E) \\
 & & \mid & p * q \mid p \oplus q \mid p \vee q \mid \ldots \\
(\textit{RelAct}) & R, G & ::= & [p] \mid p \ltimes q \mid p \propto q \mid R * R \mid \ldots
\end{array}
$$

图 5.7 插桩后的程序语言、关系式状态模型及断言语言

下面我们简要解释辅助指令对 Δ 的修改。形式化的语义可见第 5.3.4节。对于 Δ 里的每个 (U, σ_a)，辅助指令 **linself** 执行 U 中当前线程的未完成的抽象操作，相应地更新抽象对象内存 σ_a；辅助指令 **lin**(E) 则执行 U 中线程 E 的抽象操作。只有当我们确定可线性化点时，才会执行 **linself** 或 **lin**(E)。而不确定性由 **trylinself** 和 **trylin**(E) 指令引入。若在某个程序点我们不知道线程的抽象操作有没有完成，我们就记下完成和未完成两种投机情况。具体来说，设 $(U, \sigma_a) \in \Delta$ 且 U 中当前线程尚未完成抽象操作，则 **trylinself** 会向 Δ 中添加新的投机情况 (U', σ'_a)，其中 U' 中当前线程的抽象操作已完成，σ'_a 是做完该操作后的抽象对象内存。原有的 (U, σ_a) 仍然被保留。类似地，指令 **trylin**(E) 投机地执行线程 E 的抽象操作。为简单起见，我们假设 **lin**(E) 和 **trylin**(E) 中的 E 都不会使用抽象对象的变量。当我们有关于线程的抽象操作和状态的足够的知识 p 时，可以用 **commit**(p) 指令保留 Δ 中与 p 一致的投机情况，丢弃不一致的投机情况。这里 p 是状态 Σ 上的逻辑断言，我们会在下一节介绍。

5.3.2　断言

断言语言在图 5.7中定义。像 Rely-Guarantee 逻辑 [22] 那样，我们有单个状态上的断言 p 和 q，以及二元状态关系上的依赖/保证条件 R 和 G。注意这里“状态”是指图 5.7中定义的关系式状态 Σ。

图 5.8(b) 展示了状态断言的语义。我们用标准的分离逻辑断言描述具体对象内存 σ。类似第 4.3节，我们把程序变量视作资源 [39]，用 $\textit{own}(x)$ 表示具体程序拥有 x 的所有权。$E_1 \mapsto E_2$ 表示一个单地址的具体堆。

我们引入新的状态断言描述 Δ。$\boldsymbol{own}(x)$ 表示抽象程序在 Δ 的每个投机情况中都拥有 x 的所有权。不失一般性，假设抽象程序与具体实现用到的变量存储

$$f \perp g \;\; \text{iff} \;\; dom(f) \cap dom(g) = \emptyset$$

$$(s_1, h_1) \perp (s_2, h_2) \;\; \text{iff} \;\; s_1 \perp s_2 \wedge h_1 \perp h_2 \qquad (s_1, h_1) \uplus (s_2, h_2) \stackrel{\text{def}}{=} (s_1 \uplus s_2, h_1 \uplus h_2)$$

$$\Delta_1 \perp \Delta_2 \;\; \text{iff} \;\; \forall U_1, \sigma_1, U_2, \sigma_2.\, (U_1, \sigma_1) \in \Delta_1 \wedge (U_2, \sigma_2) \in \Delta_2 \implies U_1 \perp U_2 \wedge \sigma_1 \perp \sigma_2$$

$$\Delta_1 * \Delta_2 \stackrel{\text{def}}{=} \{(U_1 \uplus U_2, \sigma_1 \uplus \sigma_2) \mid (U_1, \sigma_1) \in \Delta_1 \wedge (U_2, \sigma_2) \in \Delta_2\}$$

$$\Sigma_1 * \Sigma_2 \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Delta_1 * \Delta_2), \;\; \text{if } \Sigma_1 = (\sigma_1, \Delta_1), \Sigma_2 = (\sigma_2, \Delta_2), \sigma_1 \perp \sigma_2 \text{ and } \Delta_1 \perp \Delta_2$$

$$\Sigma_1 \oplus \Sigma_2 \stackrel{\text{def}}{=} (\sigma, \Delta_1 \cup \Delta_2), \;\; \text{if } \Sigma_1 = (\sigma, \Delta_1) \text{ and } \Sigma_2 = (\sigma, \Delta_2)$$

(a) 辅助定义

$$((s,h),\Delta) \models B \qquad \text{iff } \forall U, s_a, h_a.\, (U, (s_a, h_a)) \in \Delta \implies [\![B]\!]_{s \uplus s_a} = \textbf{true}$$

$$((s,h),\Delta) \models own(x) \qquad \text{iff } (dom(s) = \{x\}) \wedge (\forall U, s_a, h_a.\, (U, (s_a, h_a)) \in \Delta \implies (s_a = \emptyset))$$

$$((s,h),\Delta) \models \boldsymbol{own}(x) \qquad \text{iff } (s = \emptyset) \wedge (\forall U, s_a, h_a.\, (U, (s_a, h_a)) \in \Delta \implies (dom(s_a) = \{x\}))$$

$$((s,h),\Delta) \models \mathsf{emp} \qquad \text{iff } (s = \emptyset) \wedge (h = \emptyset) \wedge (\Delta = \{(\emptyset, (\emptyset, \emptyset))\})$$

$$((s,h),\Delta) \models E_1 \mapsto E_2 \;\; \text{iff } \exists l, n.\, [\![E_1]\!]_s = l \wedge [\![E_2]\!]_s = n \wedge h = \{l \rightsquigarrow n\} \wedge \Delta = \{(\emptyset, (\emptyset, \emptyset))\}$$

$$\begin{aligned}((s,h),\Delta) \models E_1 \Mapsto E_2 \;\; \text{iff } & \exists s_a, h_a, l, n.\, \Delta = \{(\emptyset, (s_a, h_a))\} \wedge s = \emptyset \wedge h = \emptyset \\ & \wedge [\![E_1]\!]_{s_a} = l \wedge [\![E_2]\!]_{s_a} = n \wedge h_a = \{l \rightsquigarrow n\}\end{aligned}$$

$$\begin{aligned}((s,h),\Delta) \models E_1 \rightarrowtail (\gamma, E_2) \quad \text{iff } & \exists U, s_a, t, n.\, \Delta = \{(U, (s_a, \emptyset))\} \wedge h = \emptyset \\ & \wedge [\![E_1]\!]_{s \uplus s_a} = t \wedge [\![E_2]\!]_{s \uplus s_a} = n \wedge U = \{t \rightsquigarrow (\gamma, n)\}\end{aligned}$$

$$\begin{aligned}((s,h),\Delta) \models E_1 \rightarrowtail (\textbf{end}, E_2) \;\; \text{iff } & \exists U, s_a, t, n.\, \Delta = \{(U, (s_a, \emptyset))\} \wedge h = \emptyset \\ & \wedge [\![E_1]\!]_{s \uplus s_a} = t \wedge [\![E_2]\!]_{s \uplus s_a} = n \wedge U = \{t \rightsquigarrow (\textbf{end}, n)\}\end{aligned}$$

$$\Sigma \models p * q \;\; \text{iff} \;\; \exists \Sigma_1, \Sigma_2.\, \Sigma = \Sigma_1 * \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q$$

$$\Sigma \models p \oplus q \;\; \text{iff} \;\; \exists \Sigma_1, \Sigma_2.\, \Sigma = \Sigma_1 \oplus \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q$$

(b) 状态断言的语义

$$(\Sigma, \Sigma') \models [p] \;\; \text{iff} \;\; \Sigma \models p \wedge \Sigma = \Sigma'$$

$$(\Sigma, \Sigma') \models p \ltimes q \;\; \text{iff} \;\; \Sigma \models p \wedge \Sigma' \models q$$

$$(\Sigma, \Sigma') \models p \propto q \;\; \text{iff} \;\; \exists \Sigma_1, \Sigma_1'.\, (\Sigma = \Sigma_1 \oplus \Sigma) \wedge (\Sigma' = \Sigma_1' \oplus \Sigma) \wedge \Sigma_1 \models p \wedge \Sigma_1' \models q$$

$$\begin{aligned}&(\Sigma, \Sigma') \models R_1 * R_2 \;\; \text{iff} \\ &\quad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\, (\Sigma = \Sigma_1 * \Sigma_2) \wedge (\Sigma' = \Sigma_1' * \Sigma_2') \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2\end{aligned}$$

$$\mathsf{Id} \stackrel{\text{def}}{=} [\mathsf{true}] \qquad \mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp} \qquad \mathsf{True} \stackrel{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true}$$

$$[G]_I \stackrel{\text{def}}{=} (G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

(c) 动作断言的语义

图 5.8 断言的语义

区不相交，因此布尔断言 B 可在二者的并集上求值，从而描述具体变量与抽象变量之间的关系。$E_1 \Mapsto E_2$ 描述了 Δ 中的堆，此时 pending 线程池是空的。而 $E_1 \rightarrowtail (\gamma, E_2)$ 和 $E_1 \rightarrowtail (\mathbf{end}, E_2)$ 则描述了 pending 线程池的唯一线程 E_1。

分离合取 $p * q$ 的定义与分离逻辑中的类似，但此时是 在关系式状态 Σ 上解释的。注意此时用于定义分离合取的 Δ 上“不相交并”（disjoint union）操作（表示为 $\Delta_1 * \Delta_2$）与通常集合之间的不相交并操作不同。如图 5.8(a) 所示，$\Delta_1 * \Delta_2$ 描述的是 pending 线程池以及抽象对象内存上的不相交并。例如，下面的等式左边的 Δ 描述了线程 t_1 和 t_2 的两个投机情况（为简单起见，假设抽象对象内存为空且省略）。它可以分成等式右边的 Δ_1 和 Δ_2，二者各自描述了一个线程上的投机情况。

$$\left\{\begin{array}{l}\mathsf{t}_1 \boxed{\Upsilon_1} \\ \mathsf{t}_2 \boxed{\Upsilon_2}\end{array}, \begin{array}{l}\mathsf{t}_1 \boxed{\Upsilon_1} \\ \mathsf{t}_2 \boxed{\Upsilon_2'}\end{array}\right\} = \begin{array}{c}\left\{\mathsf{t}_1 \boxed{\Upsilon_1}\right\} \\ * \\ \left\{\mathsf{t}_2 \boxed{\Upsilon_2}, \ \mathsf{t}_2 \boxed{\Upsilon_2'}\right\}\end{array}$$

最有趣的新断言是 $p \oplus q$，它在抽象层引入了不确定性。这里 p 和 q 可以描述两种不同的投机情况。$p \oplus q$ 表示 Δ 中既包括满足 p 的投机情况，又包括满足 q 的投机情况。因此 $\oplus$ 更像是合取（conjunction），而不是析取（disjunction）。作为一个例子，我们可以把上面的等式用断言的形式表达出来：

$$\begin{aligned} & (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon_2) \oplus (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon_2') \\ \Leftrightarrow \ & \mathsf{t}_1 \rightarrowtail \Upsilon_1 * (\mathsf{t}_2 \rightarrowtail \Upsilon_2 \oplus \mathsf{t}_2 \rightarrowtail \Upsilon_2') \end{aligned}$$

依赖/保证断言描述了 Σ 上的动作（即状态转换）。图 5.8(c) 定义了它们的语义。这里我们遵照 LRG [24] 中的定义，用 $[p]$ 表示满足 p 的状态上的同一（identity）转换，用 $p \ltimes q$ 表示从满足 p 的状态到满足 q 的状态的转换。此外，我们引入一个新的断言 $p \propto q$ 描述投机的动作。它要求动作前有满足 p 的投机情况，动作后添加新的满足 q 的投机情况，而原有的投机情况都会被保留。例如，$(\mathsf{t} \rightarrowtail \Upsilon) \propto (\mathsf{t} \rightarrowtail \Upsilon')$ 表示线程 t 投机地执行操作 Υ。该动作会同时保留完成和未完成两种情况，且完成操作的结果为 Υ'。我们会在第 5.5节的例子中展示断言的更多使用。

5.3.3　推理规则

图 5.9展示了我们的逻辑的推理规则。每个逻辑判断（judgment）都以当前线程的标识号 t 为参数。当推理原子块 $\langle \widetilde{C} \rangle$ 中的代码 $\widetilde{C}$ 时，我们应用图 5.9上半部分的霍尔风格的串行规则。

$$\frac{p \Rrightarrow_{\mathsf{t}} q}{\vdash_{\mathsf{t}} \{p\}\mathbf{linself}\{q\}}\ (\text{LINSELF}) \qquad \frac{p \Rightarrow (E = E) \qquad p \Rrightarrow_E q}{\vdash_{\mathsf{t}} \{p\}\mathbf{lin}(E)\{q\}}\ (\text{LIN})$$

$$\frac{p \Rrightarrow_{\mathsf{t}} q}{\vdash_{\mathsf{t}} \{p\}\mathbf{trylinself}\{p \oplus q\}}\ (\text{TRYSELF}) \qquad \frac{p \Rightarrow (E = E) \qquad p \Rrightarrow_E q}{\vdash_{\mathsf{t}} \{p\}\mathbf{trylin}(E)\{p \oplus q\}}\ (\text{TRY})$$

$$\frac{\mathsf{SpecExact}(p) \qquad q_1|_p \Rrightarrow q_2}{\vdash_{\mathsf{t}} \{q_1\}\mathbf{commit}(p)\{q_2\}}\ (\text{COMMIT})$$

$$\frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}\mathbf{E}[\,\mathbf{return}\ E\,]\{\mathsf{t} \rightarrowtail (\mathbf{end}, E)\}}\ (\text{RET})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\ (\text{FRAME}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \vdash_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}}{\vdash_{\mathsf{t}} \{p \oplus p'\}\widetilde{C}\{q \oplus q'\}}\ (\text{SPEC-CONJ})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}\ (\text{ENV})$$

$$\frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad (p \ltimes q) \Rightarrow G * \mathsf{True} \qquad p \vee q \Rightarrow I * \mathsf{true} \qquad I \triangleright G}{[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\ (\text{ATOM})$$

$$\frac{[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\} \qquad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \qquad I \triangleright R}{R, G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\ (\text{ATOM-R})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1\{q\} \qquad R, G, I \vdash_{\mathsf{t}} \{q\}\widetilde{C}_2\{r\}}{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}_1; \widetilde{C}_2\{r\}}\ (\text{P-SEQ})$$

$$\frac{R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \mathsf{Sta}(r, R' * \mathsf{Id}) \qquad I' \triangleright \{R', G'\} \qquad r \Rightarrow I' * \mathsf{true}}{R * R', G * G', I * I' \vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\ (\text{P-FRAME})$$

图 5.9 推理规则

在 LINSELF 规则中，我们用 $p \Rrightarrow_{\mathsf{t}} q$ 表示在满足 p 的状态下执行当前线程 t 的抽象操作而得到满足 q 的状态。它的定义在图 5.10中。其中 Δ 上的转换定义为 $\Delta \rightarrow_{\mathsf{t}} \Delta'$。对于 Δ 里的每个 (U, σ)，只要线程 t 的抽象操作尚未完成，就会执行该抽象代码，相应地更新 (U, σ)。LIN 规则类似。注意我们通过条件 $p \Rightarrow (E = E)$ 来要求初始状态必须含有对 E 求值所需的变量资源。TRYSELF 规则允许当前线程 t 投机地执行抽象操作。投机的结果既包含执行抽象操作后的状态断言 q，也包含执行前的 p。TRY 规则与 TRYSELF 规则类似。

指令 $\mathbf{commit}(p)$ 提交满足 p 的投机情况。我们要求 p 满足 SpecExact，它在图 5.10中定义。简单来讲，$\mathsf{SpecExact}(p)$ 要求 p 精确地刻画一个投机情况集合，该集合中的每个投机情况描述的是同一些线程和同样“大小”的对象内存

$$\frac{U(\mathsf{t}) = (\mathbf{end}, n)}{(U,\sigma) \dashrightarrow_{\mathsf{t}} (U,\sigma)} \qquad \frac{\begin{array}{c} U(\mathsf{t}) = ((x, \langle C\rangle), n) \\ (\langle C\rangle, (s \uplus \{x \rightsquigarrow n\}, h)) \longrightarrow_{\mathsf{t}} (\mathbf{fret}(n'), (s', h')) \end{array}}{(U,(s,h)) \dashrightarrow_{\mathsf{t}} (U\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, (s' \backslash \{x\}, h'))}$$

$$\frac{}{\emptyset \rightarrow_{\mathsf{t}} \emptyset} \qquad \frac{(U,\sigma) \dashrightarrow_{\mathsf{t}} (U',\sigma') \qquad \Delta \rightarrow_{\mathsf{t}} \Delta'}{\{(U,\sigma)\} \uplus \Delta \ \rightarrow_{\mathsf{t}} \ \{(U',\sigma')\} \cup \Delta'}$$

$$\begin{array}{l} p \Rrightarrow_E q \ \text{ iff} \\ \quad \forall s, h, \Delta, \mathsf{t}.\ (((s,h),\Delta) \models p) \wedge ([\![E]\!]_s = \mathsf{t}) \implies \exists \Delta'.\ (\Delta \rightarrow_{\mathsf{t}} \Delta') \wedge ((s,h),\Delta') \models q \end{array}$$

$$\begin{array}{l} \Delta|_{\Delta_p} \stackrel{\text{def}}{=} \\ \quad \{(U,\sigma) \mid (U,\sigma) \in \Delta \wedge \exists U_p, \sigma_p, U_1, \sigma_1.\ (U_p,\sigma_p) \in \Delta_p \wedge U = U_p \uplus U_1 \wedge \sigma = \sigma_p \uplus \sigma_1\} \\ (\sigma,\Delta)|_p = \Delta' \ \text{ iff } \ \exists \sigma_p, \Delta_p, \sigma_1.\ ((\sigma_p,\Delta_p) \models p) \wedge (\sigma = \sigma_p \uplus \sigma_1) \wedge (\Delta|_{\Delta_p} = \Delta') \\ q_1|_p \Rrightarrow q_2 \ \text{ iff } \ \forall \sigma, \Delta.\ (\sigma,\Delta) \models q_1 \implies \exists \Delta'.\ ((\sigma,\Delta)|_p = \Delta') \wedge (\sigma,\Delta') \models q_2 \end{array}$$

$$\begin{array}{l} \mathsf{DomExact}(\Delta) \ \text{ iff } \ \forall U, \sigma, U', \sigma'.\ (U,\sigma) \in \Delta \wedge (U',\sigma') \in \Delta \implies \\ \qquad\qquad\qquad (\mathit{dom}(U) = \mathit{dom}(U')) \wedge (\mathit{dom}(\sigma) = \mathit{dom}(\sigma')) \\ \mathsf{SpecExact}(p) \ \text{ iff } \ \forall \Delta, \Delta'.\ ((\_,\Delta) \models p) \wedge ((\_,\Delta') \models p) \implies \\ \qquad\qquad\qquad (\Delta = \Delta') \wedge \mathsf{DomExact}(\Delta) \\ \mathsf{Precise}(p) \ \text{ iff } \ \forall \sigma_1, \Delta_1, \sigma_2, \Delta_2, \sigma_1', \Delta_1', \sigma_2', \Delta_2'. \\ \qquad\qquad ((\sigma_1 \uplus \sigma_2 = \sigma_1' \uplus \sigma_2') \wedge ((\sigma_1,\_) \models p) \wedge ((\sigma_2,\_) \models p) \implies (\sigma_1 = \sigma_2)) \wedge \\ \qquad\qquad ((\Delta_1 * \Delta_2 = \Delta_1' * \Delta_2') \wedge ((\_,\Delta_1) \models p) \wedge ((\_,\Delta_1) \models p) \implies (\Delta_1 = \Delta_2)) \\ \mathsf{Sta}(p,R) \ \text{ iff } \ \forall \Sigma, \Sigma'.\ (\Sigma \models p) \wedge ((\Sigma,\Sigma') \models R) \implies \Sigma' \models p \\ I \triangleright R \ \text{ iff } \ ([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I) \end{array}$$

图 5.10 推理规则的辅助定义

（DomExact）。例如下面的 p_1 满足 SpecExact，而 p_2 和 p_3 不满足：

$$\begin{array}{rcl} p_1 & \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \ \oplus \ \mathsf{t} \rightarrowtail (\mathbf{end}, n') \\ p_2 & \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \ \vee \ \mathsf{t} \rightarrowtail (\mathbf{end}, n') \\ p_3 & \stackrel{\text{def}}{=} & \mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \ \oplus \ \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2) \end{array}$$

在第 5.5节的所有例子中，**commit**(p) 的断言 p 都刻画唯一的投机情况，此时 SpecExact(p) 平凡地成立。在 COMMIT 规则中，我们用 $q_1|_p \Rrightarrow q_2$ 从前条件 q_1 中“过滤”出符合 p 的投机情况 q_2。其定义（在图 5.10中）的核心是 Δ 上的过滤器 $\Delta|_{\Delta_p}$。这里 Δ_p 是 p 代表的投机情况集合，它描述了确定定义域的线程和抽象内存资源。Δ 可以包含额外的线程或抽象内存资源。例如，下面的 Δ 刻画了两个线程 t_1 和 t_2，但 Δ_p 可以只提到一个线程 t_1。

$$\Delta : \left\{ \begin{array}{c|c} \mathsf{t}_1 & (\gamma_1, n_1) \\ \hline \mathsf{t}_2 & (\gamma_2, n_2) \end{array}, \quad \begin{array}{c|c} \mathsf{t}_1 & (\mathbf{end}, n_1') \\ \hline \mathsf{t}_2 & (\mathbf{end}, n_2') \end{array} \right\}$$

设 p 为 $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)$，则经过过滤器 $\Delta|_{\Delta_p}$ 后，仅有左边的投机情况会保留。也就是说，下述逻辑判断成立：

$$\vdash_{\mathsf{t}} \ \{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\}$$
$$\mathbf{commit}(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1))$$
$$\{\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)\}$$

而若 p 为 $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')$，则过滤器 $\Delta|_{\Delta_p}$ 会同时保留两个投机情况，因此下述逻辑判断成立：

$$\vdash_{\mathsf{t}} \ \{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\}$$
$$\mathbf{commit}(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1'))$$
$$\{(\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) * \mathsf{t}_2 \rightarrowtail (\gamma_2, n_2)) \oplus (\mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1') * \mathsf{t}_2 \rightarrowtail (\mathbf{end}, n_2'))\}$$

RET 规则用于推理对象方法的返回，此时我们必须知道当前线程的抽象操作已经完成。我们还支持分离逻辑中的标准的 FRAME 规则，用于局部推理。此外，SPEC-CONJ 规则让我们能够分开考虑各个投机情况下的验证。它很像传统霍尔逻辑中的合取规则。

图 5.9下半部分的规则用于 Rely-Guarantee 风格的并发推理，它们与并发程序逻辑 LRG [24] 中的规则完全一致。像 LRG 那样，我们需要一个精确的不变式（precise invariant）I 来刻画共享资源的边界。当共享资源是空的（即 $\widetilde{C}$ 仅使用局部资源）时，我们应用 ENV 规则把 $\widetilde{C}$ 当作串行程序来验证。这里 Emp 表示空资源上的状态转换（见图 5.8中定义）。ATOM 规则让我们将原子块中的代码当作串行程序来验证，条件是共享资源上的状态转换满足保证条件 G，且 G 被不变式 I 所束缚（fence）。束缚 $I \triangleright G$ 在图 5.10中定义。在这条规则中我们假设环境不会修改共享资源，因此依赖条件为 $[I]$，即 I 上的同一（identity）转换。为了支持一般的（非同一的）环境 R，我们可以再应用 ATOM-R 规则。它要求 R 被 I 所束缚，而且前后条件 p 和 q 在 $R * \mathsf{Id}$ 下稳定，这里 R 描述了环境对共享资源的修改，Id 是局部资源上的同一转换（环境不会修改线程的局部资源）。稳定性 $\mathsf{Sta}(p, R)$ 在图 5.10中定义。我们还展示了在并发环境下推理顺序组合的 P-SEQ 规则以及支持局部推理的 P-FRAME 规则。它们都与 LRG 中的对应规则一致，但要注意这里的断言都是在关系式状态 Σ 上解释的。

与客户端程序的验证链接起来　上述逻辑是用于验证并发对象的实现的，但其实它是通用并发程序逻辑 LRG 的一个扩展，因此也能够用于客户端程序的验证。虽然这里我们没有展示并发组合的规则，但它实际上支持 LRG 中的所有推理规则（包括并发组合规则）。此外，在第 5.4节我们将会看到，该逻辑能够保证上下文精化。因此，验证程序 W 的时候，我们可以将对象具体实现替换为对应的抽象操作，转而验证替换后的程序。注意到替换后的程序把对象的具体表示和方法的实现细节都抽象掉了，也就是说，这一验证方法做到了对象实现的

$$\frac{(C,\sigma) \longrightarrow_{\mathsf{t}} (C',\sigma') \qquad C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]}{(C,(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (C',(\sigma',\Delta))}$$

$$\frac{[\![E]\!]_s = n \qquad \forall U.\,(U,\_) \in \Delta \implies U(\mathsf{t}) = (\mathbf{end},n)}{(\mathbf{E}[\,\mathbf{return}\ E\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip},((s,h),\Delta))}$$

$$\frac{\Delta \rightarrow_{\mathsf{t}} \Delta'}{(\mathbf{E}[\,\mathbf{linself}\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta'))}$$

$$\frac{[\![E]\!]_s = \mathsf{t}' \qquad \Delta \rightarrow_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{lin}(E)\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],((s,h),\Delta'))}$$

$$\frac{\Delta \rightarrow_{\mathsf{t}} \Delta'}{(\mathbf{E}[\,\mathbf{trylinself}\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta\cup\Delta'))}$$

$$\frac{[\![E]\!]_s = \mathsf{t}' \qquad \Delta \rightarrow_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{trylin}(E)\,],((s,h),\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],((s,h),\Delta\cup\Delta'))}$$

$$\frac{\mathsf{SpecExact}(p) \qquad (\sigma,\Delta)|_p = \Delta'}{(\mathbf{E}[\,\mathbf{commit}(p)\,],(\sigma,\Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(\sigma,\Delta'))}$$

$$\frac{(\widetilde{C},\Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}',\Sigma')}{(\widetilde{C},\Sigma) \xhookrightarrow{R}_{\mathsf{t}} (\widetilde{C}',\Sigma')} \qquad \frac{(\Sigma,\Sigma') \models R}{(\widetilde{C},\Sigma) \xhookrightarrow{R}_{\mathsf{t}} (\widetilde{C},\Sigma')}$$

图 5.11 插桩后的程序在关系式状态下的操作语义

“分离和信息隐藏”（separation and information hiding）[51]，但同时又给我们保留了足够的信息（即抽象对象）来验证并发客户端程序中的方法调用。

5.3.4　语义和部分正确性

我们首先展示插桩后的代码 $\widetilde{C}$ 在关系式状态 Σ 下的操作语义，如图 5.11。线程 t 的一步执行表示为 $(\widetilde{C},\Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}',\Sigma')$。当执行到 **return** E 指令时（见图 5.11中的第二条规则），线程 t 的抽象操作必须确定已经完成。具体来讲，在 Δ 的每个投机情况 U 下，$U(\mathsf{t})$ 都必须是 **end**，且返回值均为 E。辅助指令的含义都已经在前面解释过。我们用 $\Delta \rightarrow_{\mathsf{t}} \Delta'$ 刻画线程 t 执行其抽象操作对 Δ 的改变。**commit**(p) 的语义要求 p 满足 SpecExact，它用 $(\sigma,\Delta)|_p = \Delta'$ 过滤出符合 p 的投机情况。这些辅助定义都在图 5.10中，且已经在前面介绍过。

在单线程的操作语义的基础上，我们可以定义 $(\widetilde{C},\Sigma) \xhookrightarrow{R}_{\mathsf{t}} (\widetilde{C},\Sigma)$，描述线程 t 在环境 R 的干扰下的行为。

插桩的语义保持　容易看出新引入的辅助指令不会修改物理状态 σ，也不会影响程序的控制流。因此，插桩辅助指令并不会改变原程序的行为，除非这些辅

助指令被插入了错误的程序点而使整个程序无法继续执行下去，但是这种情况可以通过验证来排除。

程序逻辑的部分正确性　像 LRG [24] 那样，我们可以定义逻辑判断的语义 $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$，表达 $\widetilde{C}$ 在前后条件、依赖/保证条件下的部分正确性。我们首先定义 $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$，用于串行推理的逻辑判断的语义。

定义 5.7 (串行逻辑判断的语义). $\models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ 成立，当且仅当对于任意 Σ，如果 $\Sigma \models p$，那么下述成立：

1. 对于任意 Σ'，如果 $(\widetilde{C}, \Sigma) \longrightarrow_{\mathsf{t}}^{*} (\mathbf{skip}, \Sigma')$，那么 $\Sigma' \models q$ 成立；
2. $(\widetilde{C}, \Sigma) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$ 成立。

定义 5.8 (并发逻辑判断的语义). $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ 成立，当且仅当对于任意 Σ，如果 $\Sigma \models p$，那么下述成立：

1. 对于任意 Σ'，如果 $(\widetilde{C}, \Sigma) \xrightarrow{R*\mathsf{Id}}_{\mathsf{t}}^{*} (\mathbf{skip}, \Sigma')$，那么 $\Sigma' \models q$ 成立；
2. 对于任意 n, $(\widetilde{C}, \Sigma, R * \mathsf{Id})$ $\mathsf{guarantees}_{\mathsf{t}}^{n}$ $(G * \mathsf{True})$ 成立。

其中，性质 $(\widetilde{C}, \Sigma, R)$ $\mathsf{guarantees}_{\mathsf{t}}^{n}$ G 如下归纳定义：

1. $(\widetilde{C}, \Sigma, R)$ $\mathsf{guarantees}_{\mathsf{t}}^{0}$ G 恒成立；
2. $(\widetilde{C}, \Sigma, R)$ $\mathsf{guarantees}_{\mathsf{t}}^{k+1}$ G 当且仅当
   - *(a)* $(\widetilde{C}, \Sigma) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$ 成立；
   - *(b)* 对于任意 Σ'，如果 $(\Sigma, \Sigma') \models R$, 那么 $(\widetilde{C}, \Sigma', R)$ $\mathsf{guarantees}_{\mathsf{t}}^{k}$ G 成立；
   - *(c)* 对于任意 $\widetilde{C}'$ 和 Σ'，如果 $(\widetilde{C}, \Sigma) \longrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$，那么 $(\Sigma, \Sigma') \models G$ 且 $(\widetilde{C}', \Sigma', R)$ $\mathsf{guarantees}_{\mathsf{t}}^{k}$ G 成立。

简单来讲，$(\widetilde{C}, \Sigma, R)$ $\mathsf{guarantees}_{\mathsf{t}}^{n}$ G 是指，若线程 t 的代码 $\widetilde{C}$ 从初始状态 Σ 开始在环境 R 的干扰下执行 n 步，则 $\widetilde{C}$ 不会出错且每一步的状态转换都满足 G。在此基础上可定义逻辑判断的语义：当初始状态满足 p 时，$\widetilde{C}$ 在环境 R 下的所有执行都不会出错，且每一步都满足 G，执行终止时的状态满足 q。我们可以证明我们的程序逻辑保证部分正确性，即逻辑判断蕴涵上述语义。

定理 5.9 (逻辑的部分正确性). 如果 $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$，那么 $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$。

下一节我们介绍该程序逻辑的一种更强的可靠性保证：对于线性一致性的可靠性。

5.4　保证程序逻辑可靠性的模拟关系

直观上，我们的程序逻辑将对象的具体实现和对应的抽象操作联系起来。本节我们将这一直观感受形式化，证明该逻辑确实保证了对象的线性一致性。证明过程分为以下几步。首先，我们提出一种新的、建立在具体实现和抽象操作之间的、基于依赖 -保证的 forward-backward 模拟关系。我们证明该模拟关系具有可组合性，蕴涵具体实现和抽象操作之间的上下文精化（定义 5.5）。然后我们证明我们的程序逻辑确立了该模拟关系，因此它也保证了上下文精化。最后由上下文精化和线性一致性之间的等价（定理 5.6）可知，我们的逻辑对于线性一致性是可靠的。

下面我们先定义基于依赖 -保证的 forward-backward 模拟关系。它扩展了第 2 章介绍的模拟关系 RGSim，以支持帮助机制和投机机制。

定义 5.10 (方法实现和抽象操作之间的新模拟关系)**.** $R, G, I \models_{\mathsf{t}} (x, C) \preceq_p \gamma$ 当且仅当

$$\forall \sigma, \Delta.\ (\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, x) * p) \implies R, G, I \models_{\mathsf{t}} (C; \mathbf{noret}, \sigma) \preceq_p \Delta\,.$$

若 $R, G, I \models_{\mathsf{t}} (C, \sigma) \preceq_p \Delta$，则 $(\sigma, \Delta) \models I * \mathsf{true}$ 且下述全部成立：

1. 若 $C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$，则
   (a) 对于任意 C'，σ''，σ_F 和 Δ_F，若 $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$ 且 $\Delta \perp \Delta_F$，则存在 σ' 和 Δ' 使得 $\sigma'' = \sigma' \uplus \sigma_F$，$(\Delta * \Delta_F) \rightrightarrows (\Delta' * \Delta_F)$，$((\sigma, \Delta), (\sigma', \Delta')) \models (G * \mathsf{True})$ 和 $R, G, I \models_{\mathsf{t}} (C', \sigma') \preceq_p \Delta'$ 成立；
   (b) 对于任意 σ_F，$(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$ 成立；
2. 对于任意 σ' 和 Δ'，若 $((\sigma, \Delta), (\sigma', \Delta')) \models (R * \mathsf{Id})$，则 $R, G, I \models_{\mathsf{t}} (C, \sigma') \preceq_p \Delta'$ 成立；
3. 若 $C = \mathbf{E}[\,\mathbf{return}\ E\,]$ 且 $\sigma = (s, h)$，则存在 n' 使得 $[\![E]\!]_s = n'$ 且 $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\mathbf{end}, n') * \mathit{own}(x) * p)$ 成立。

像 RGSim 那样，$R, G, I \models_{\mathsf{t}} (x, C) \preceq_p \gamma$ 建立了在并发环境的干扰（由 R 和 G 刻画）下具体实现 C 与抽象操作 γ 之间的模拟关系。它将具体代码 C 的执行同某个 Δ 的投机执行联系起来。这个 Δ 既可以描述当前线程 t 的抽象操作，也可以描述 t 可能帮助的其他线程的抽象操作。初始 t 的抽象操作是 γ，参数与具体实现的参数 x 相等。其他线程的抽象操作可以从前条件 p 中知道。

定义 5.10的第 1 条要求具体代码 C 的每一步都应是安全的，且对应于 Δ 的若干步。其中，$\Delta \rightrightarrows \Delta'$ 如下定义：

$$\Delta \rightrightarrows \Delta' \ \text{ iff } \ \forall U', \sigma'.\ (U', \sigma') \in \Delta' \implies \exists U, \sigma.\ (U, \sigma) \in \Delta \wedge (U, \sigma) \dashrightarrow^* (U', \sigma')\,,$$

其中，$(U, \sigma) \dashrightarrow (U', \sigma') \ \stackrel{\text{def}}{=}\ \exists \mathsf{t}.\ (U, \sigma) \dashrightarrow_{\mathsf{t}} (U', \sigma')$

且$(U, \sigma) \dashrightarrow_{\mathsf{t}} (U', \sigma')$定义在图 5.10 中

简单来讲，$\Delta \rightrightarrows \Delta'$ 要求 Δ' 中的所有 (U', σ') 都是从 Δ “可达” 的。这允许我们执行 Δ 中某个线程 t'（可以是当前线程 t）的抽象操作，或者丢弃 Δ 中的某些 (U, σ)。前者的效果可以由 **lin**(t') 或 **trylin**(t') 的一步产生，我们可以选择是否保留线程 t' 的抽象操作。后者可以视为 **commit** 的一步，它丢弃了错误的投机情况。

受 Vafeiadis 的工作 [52] 的启发，我们在定义 5.10中直接地描述程序执行对任意的额外资源 σ_F 和 Δ_F 的影响。直观上，σ_F 和 Δ_F 代表其他线程拥有的局部资源，当前线程在具体层和抽象层的执行都不能修改它们。

我们还要求具体层和抽象层的对应执行满足保证条件 $G * \mathsf{True}$，其中共享资源上的转换应满足 G，局部资源上的转换可以是任意的（满足 True）。对称地，定义 5.10的第 2 条要求在环境 $R * \mathsf{Id}$ 的干扰下该模拟关系仍能保持。其中环境对共享资源的修改满足 R，而环境不能修改当前线程的局部资源，因此局部资源上的转换是同一的（满足 Id）。

最后，当方法返回时，定义 5.10的第 3 条要求当前线程 t 必须已经完成其抽象操作，且两层的返回值相等。

同 RGSim 一样，新的模拟关系也具有可组合性（compositionality）。利用其可组合性，我们可以证明它保证了具体实现和抽象操作之间的上下文精化，如下面的引理所示。

引理 5.11 (模拟关系蕴涵上下文精化)**.** *对于任意 Π，Π_A 和 φ，若存在 R，G，I 和 p 使得对于任意 t，下述成立：*

1. *对于任意 f，设 $\Pi(f) = (x, C)$，则我们有 $R_{\mathsf{t}}, G_{\mathsf{t}}, I \models_{\mathsf{t}} \Pi(f) \preceq_{p_{\mathsf{t}}} \Pi_A(f)$ 以及 $x \notin dom(I)$ 成立；*
2. *$R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$，$I \triangleright \{R_{\mathsf{t}}, G_{\mathsf{t}}\}$，$p_{\mathsf{t}} \Rightarrow I$，及 $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$；*
3. *$\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$；*

则 $\Pi \sqsubseteq_{\varphi} \Pi_A$ 成立。

这里我们用 $x \notin dom(I)$ 表示形参 x 不是共享资源。$\lfloor \varphi \rfloor$ 将 φ 提升为状态断言：$\lfloor \varphi \rfloor \stackrel{\text{def}}{=} \{(\sigma, \{(\emptyset, \sigma_a)\}) \mid \varphi(\sigma) = \sigma_a\}$。

引理 5.11告诉我们，要证明上下文精化 $\Pi \sqsubseteq_{\varphi} \Pi_A$，可以证明对于每个方法 f，模拟关系 $R_{\mathsf{t}}, G_{\mathsf{t}}, I \models_{\mathsf{t}} \Pi(f) \preceq_{p_{\mathsf{t}}} \Pi_A(f)$ 成立。其中 R，G 和 p 定义在由不变式 I 束缚的共享状态上，满足干涉约束 $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$。证明该引理的关键是证明模拟关系具有可组合性。可组合性的证明过程与 RGSim 的类似，但 此时我们需要考虑线程和环境之间的帮助机制，以及抽象操作的投机执行。具体的证明过程可见技术报告 [53]。

$$
\begin{array}{lll}
\mathsf{Er}(\textbf{linself}) \stackrel{\text{def}}{=} \textbf{skip} & \mathsf{Er}(\textbf{trylinself}) \stackrel{\text{def}}{=} \textbf{skip} & \mathsf{Er}(\textbf{lin}(E)) \stackrel{\text{def}}{=} \textbf{skip} \\
\mathsf{Er}(\textbf{trylin}(E)) \stackrel{\text{def}}{=} \textbf{skip} & \mathsf{Er}(\textbf{commit}(p)) \stackrel{\text{def}}{=} \textbf{skip} & \mathsf{Er}(C) \stackrel{\text{def}}{=} C \\
\mathsf{Er}(\langle \widetilde{C} \rangle) \stackrel{\text{def}}{=} \langle\, \mathsf{Er}(\widetilde{C})\, \rangle & \mathsf{Er}(\widetilde{C}_1; \widetilde{C}_2) \stackrel{\text{def}}{=} \mathsf{Er}(\widetilde{C}_1);\, \mathsf{Er}(\widetilde{C}_2) & \\
\mathsf{Er}(\textbf{if}\,(B)\,\widetilde{C}_1\,\textbf{else}\,\widetilde{C}_2) \stackrel{\text{def}}{=} \textbf{if}\,(B)\ \mathsf{Er}(\widetilde{C}_1)\ \textbf{else}\ \mathsf{Er}(\widetilde{C}_2) & & \\
\mathsf{Er}(\textbf{while}\,(B)\{\widetilde{C}\}) \stackrel{\text{def}}{=} \textbf{while}\,(B)\{\,\mathsf{Er}(\widetilde{C})\,\} & &
\end{array}
$$

图 5.12 辅助代码的擦除

引理 5.12 (逻辑保证模拟关系)**.** *对于任意* t，x，C，γ，R，G，I *和* p，*若存在* $\widetilde{C}$ *使得* $\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$ *且*

$$R, G, I \vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, x) * p\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\textbf{end}, \_) * \mathit{own}(x) * p\}$$

成立，则 $R, G, I \models_{\mathsf{t}} (x, C) \preceq_p \gamma$ *成立。*

这里我们用 $\mathsf{Er}(\widetilde{C})$ 擦除（erase）$\widetilde{C}$ 中插桩的辅助指令，它在图 5.12中定义。引理 5.12告诉我们，用我们的程序逻辑验证 $\widetilde{C}$ 就可以得到插桩前的具体实现和对应抽象操作之间的模拟关系。该引理的证明基础是逻辑能够保证部分正确性（定理 5.9），即 $\widetilde{C}$ 在关系式状态下的执行是安全的。然后我们将关系式状态下的语义（图 5.11）分别投影到具体状态下的语义（图 5.5）以及投机执行 $\Delta \Rightarrow \Delta'$，从而建立具体实现与抽象操作之间的模拟关系。

最后，由引理 5.11 和引理 5.12可得下面的可靠性定理，它保证了我们的程序逻辑可用于验证线性一致性。

定理 5.13 (逻辑的可靠性)**.** *对于任意* Π，Π_A *和* φ，*若存在* R，G，I *和* p *使得对于任意* t，*下述成立：*

1. *对于任意* f，*若* $\Pi(f) = (x, C)$，*则存在* $\widetilde{C}$ *使得*

$$R_{\mathsf{t}}, G_{\mathsf{t}}, I \vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\Pi_A(f), x) * p_{\mathsf{t}}\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\textbf{end}, \_) * \mathit{own}(x) * p_{\mathsf{t}}\}\,,$$

   $\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$，*以及* $x \notin \mathit{dom}(I)$ *成立；*

2. $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$，$p_{\mathsf{t}} \Rightarrow I$，*及* $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$；

3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$；

则 $\Pi \sqsubseteq_{\varphi} \Pi_A$ *成立，因此* $\Pi \preceq_{\varphi} \Pi_A$ *也成立。*

对象	帮助机制	未来依赖	Java 包	HS 教科书
Treiber 栈 [10]				√
HSY 栈 [18]	√			√
MS 双锁队列 [54]				√
MS 无锁队列 [54]		√	√	√
DGLM 无锁队列 [49]		√		
锁耦合的链表 [3]				√
乐观链表 [3]				√
Heller 等人的懒惰链表 [19]	√	√		√
Harris-Michael 无锁链表 [55, 56]	√	√	√	√
一对数据的快照 [45]		√		
CCAS [17]	√	√		
RDCSS [46]	√	√		

表 5.1 用我们的程序逻辑验证过的算法

5.5　应用举例

我们的程序逻辑是一种验证线性一致性的有效手段。如表 5.1所示，我们已经验证了 12 个并发对象算法，包括 2 个栈算法、3 个队列算法、4 个链表算法，以及 3 个与原子读写有关的算法。表 5.1总结了这些并发对象的特性，例如可线性化点（LP）是否可能在别的线程中（**帮助机制**栏）或者依赖未来的行为（**未来依赖**栏）。某些算法已经在 Java 并发包 `java.util.concurrent` 中使用（**Java 包**栏）。最后一栏（**HS 教科书**栏）记录算法是否出现在 Herlihy 和 Shavit 撰写的介绍并发算法的经典教科书 [3] 中。事实上，我们验证的这 12 个并发对象已经囊括了书中绝大多数栈、队列及链表实现。从表中可以看出，我们的逻辑既支持 LP 固定的简单并发对象，也支持 LP 不固定的复杂算法。尽管这里的许多例子都可以用其他方法验证，但我们的方法是第一个已证明可靠性且支持所有这些对象的程序逻辑。这些例子的完整证明过程可见技术报告 [53]。

一般来讲，我们验证对象线性一致性的过程分为以下两步。首先，我们在对象实现代码中插桩辅助指令，如 **linself**，**trylin**(E) 和 **commit**(p) 等。根据算法的原理，通常不难找到插桩的合适程序点。然后，我们按照定理 5.13的要求写下断言，用我们的逻辑规则（图 5.9）推理插桩后的代码。整个推理过程就像在 LRG 中验证并发程序的部分正确性一样，对辅助指令的处理通常并不会增加验证的难度。下面我们验证三个有代表性的算法，展示我们的程序逻辑的应用。这三个例子是：一对数据的快照，MS 无锁队列，以及 CCAS 算法。

5.5.1　一对数据的快照

我们已经在第 5.1.3节介绍过，对于一对数据快照（pair snapshot）的实现，其 LP 的位置依赖未来的行为。图 5.13展示了`readPair` 方法的证明。设当前线

```
 1 readPair(int i, j) {
     {I * (cid ↣ (γ, (i, j)))}
 2   local a, b, v, w, done := false;
     {((¬done) * I * (cid ↣ (γ, (i, j)) ⊕ true)) ∨ (done * I * (cid ↣ (end, (a, b))))}
 3   while(!done) {
       {(¬done) * I * (cid ↣ (γ, (i, j)) ⊕ true)}
 4     < a := m[i].d; v := m[i].v; >
       {∃v'. (¬done) * (I ∧ readCell(i, a, v; v')) * (cid ↣ (γ, (i, j)) ⊕ true)}
 5     < b := m[j].d; w := m[j].v; trylinself; >
       {∃v'. (¬done) * (I ∧ readCell(i, a, v; v') ∧ readCell(j, b, w; _)) * afterTry}
 6     if (v = m[i].v) {
         {I * (cid ↣ (end, (a, b)) ⊕ true)}
 7       commit(cid ↣ (end, (a, b)));
         {I * (cid ↣ (end, (a, b)))}
 8       done := true;
 9     }
10   }
     {I * (cid ↣ (end, (a, b)))}
11   return (a, b);
     {I * (cid ↣ (end, (a, b)))}
12 }
```

Auxiliary definitions:

$\mathsf{readCell}(i, d, v; v') \stackrel{\text{def}}{=} (\mathsf{cell}(i, d, v) \vee (\mathsf{cell}(i, \_, v') \wedge v < v')) * \mathsf{true}$

$\mathsf{absRes}(a, b; v, v') \stackrel{\text{def}}{=} (\mathtt{cid} \rightarrowtail (\mathbf{end}, (a, b)) \wedge v \leq v') \vee (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\_, b)) \wedge v < v')$

$\mathsf{afterTry} \stackrel{\text{def}}{=} \mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{absRes}(\mathtt{a}, \mathtt{b}; \mathtt{v}, v') \oplus \mathsf{true}$

图 5.13 `readPair` 方法实现的证明

程为 `cid`，对应的抽象操作为 γ，它在抽象层原子地读单元 `i` 和 `j`。

首先，我们插桩 **trylinself** 和 **commit** 指令，见图 5.13中的高亮代码。**commit** 指令要求，如果第 6 行的确认成功，那么 $\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b}))$ 必须是一种可能的投机情况。这意味着我们必须在合适的程序点插入 **trylinself**。在图 5.13中，它被插入于第 5 行，因为第 5 行是唯一的能使抽象操作 γ 的执行返回 $(\mathtt{a}, \mathtt{b})$ 的程序点。另一方面，它不能被替换为 **linself**，因为如果之后第 6 行的确认失败，我们就需要重新执行抽象操作 γ。

接着，我们定义不变式 I，以及依赖/保证条件 R 和 G。如下所示，不变式 I 将具体对象的每个内存单元 (d, v) 映射到抽象对象的一个值为 d 的单元。

$$I \stackrel{\text{def}}{=} \circledast_{i \in [1..\mathtt{size}]}.(\exists d, v.\ \mathsf{cell}(i, d, v))$$

$$\text{其中，}\ \mathsf{cell}(i, d, v) \stackrel{\text{def}}{=} (\mathtt{m}[i] \mapsto (d, v)) * (\mathtt{m}[i] \Mapsto d))$$

每个线程都保证修改数据的同时会增加版本号，因此我们如下定义 G：

$$G \stackrel{\text{def}}{=} [\mathit{Write}]_I \qquad \mathit{Write} \stackrel{\text{def}}{=} \exists i, v.\ \mathsf{cell}(i, \_, v) \ltimes \mathsf{cell}(i, \_, v+1)$$

这里我们用 $[G]_I$ 作为 $(G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$ 的简写（定义在图 5.8(c) 中）。依赖条件 R 与保证条件 G 相同。

```
 1 enq(v) {
 2  local x, t, s, b;
 3  x := cons(v, null);
 4  while (true) {
 5   t := Tail; s := t.next;
 6   if (t = Tail) {
 7    if (s = null) {
 8      b:=cas(&(t.next),s,x);
 9      if (b) {
10        cas(&Tail, t, x);
11        return; }
12    }else cas(&Tail, t, s);
13   }
14  }
15 }
```

```
16 deq() {
17  local h, t, s, v, b;
18  while (true) {
19   h := Head; t := Tail;
20   s := h.next;
21   if (h = Head)
22    if (h = t) {
23      if (s = null)
24        return EMPTY;
25      cas(&Tail, t, s);
26    }else {
27      v := s.val;
28      b:=cas(&Head,h,s);
29      if(b) return v; }
30  } }
```

图 5.14 MS 无锁队列的实现代码

然后，如图 5.13所示，我们写下前后条件，用我们的逻辑规则推理插桩后的代码。注意循环不变式（第 2 行与第 3 行代码之间的断言）在 $\neg$done 的情况下允许 cid $\rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus$ true，其含义为，线程 cid 可能尚未完成抽象操作，或者虽然投机地完成了但需要重做。

这个算法的readPair 方法不会修改并发对象内存。这意味着，无论当前线程线性化该方法多少次，都不会对并发的其他线程造成影响。在第 5.5.3节我们会验证一个 LP 依赖未来且会修改对象内存的算法，那时我们仍然可以“多次线性化”一个方法，而不会产生副作用。

5.5.2　MS 无锁队列

著名的 MS 无锁队列算法 [54] 也具有依赖未来的 LP。图 5.14 展示了它的代码。该算法用一个单链表实现队列，该链表具有头指针 Head 和尾指针 Tail。Head 永远指向链表的第一个节点，这个节点是一个哨兵（sentinel），不记录队列数据。Tail 指向的要么是链表的最后一个节点，要么是倒数第二个节点。入队方法enq 在链表的尾部添加一个新节点，并推进 Tail 指针。出队方法deq 将哨兵节点替换为它的下一个节点，并返回新的哨兵节点里的数据。如果链表仅含有哨兵节点，意味着队列是空的，那么deq 方法返回 EMPTY。

对于enq 方法，链接新节点和推进 Tail 指针不是原子完成的，因此可能出现 Tail 指针落后于链表最后一个节点的情况。该算法采取帮助机制来推进落后的 Tail 指针。每个线程在做自己的操作之前，会先尝试帮助只完成一半enq 的线程推进 Tail 指针（见图 5.14的第 12 行和第 25 行）。但是注意该帮助机制并不影响enq 方法的 LP。当第 8 行的cas 成功时，新的节点在链表队列中已经

可见，此时就是enq 方法的 LP。之后更新 Tail 指针并不会再改变对应的抽象队列。因此，我们如下插桩第 8 行以验证enq 的实现：

```
< b := cas(&(t.next), s, x); if (b) linself; >
```

另一方面，该算法会检查 Head 和 Tail 指针（见图 5.14中第 6 行和第 21 行），确保自从上次读取它们（第 5 行和第 19 行）后环境还未曾改变它们。若检查不通过，则重新开始循环。这种检查可以提高算法的运行效率，但却使得deq 方法在空队列情况下的 LP 依赖未来的执行。当队列为空时，deq 方法的 LP 应当在第 20 行，但条件是确实会在这一轮循环返回 EMPTY。直观上，当我们在第 20 行读到h.next 为 null 时，我们知道此时对应的抽象队列一定是空的，但我们不知道deq 方法是否会在这一轮循环返回。如果之后第 21 行对 Head 指针的检查不通过，就要重新开始循环，那么刚才执行的第 20 行就不是 LP。我们可以通过插桩 try-commit 指令来处理这种依赖未来的 LP。在第 20 行我们插入 **trylinself**，如下：

```
< s := h.next; if (h = t && s = null) trylinself; >
```

在方法返回 EMPTY（第 24 行）之前，我们插入 commit(cid ↣ (**end**, EMPTY))，提交已完成抽象操作的投机情况。而当我们知道必须继续循环时，可以提交原始的 DEQ 操作，即在循环体的末尾（第 30 行前）插入 commit(cid ↣ DEQ)。

当队列不空时，deq 方法的 LP 可以静态确定在第 28 行cas 成功的时候。因此我们可以在第 28 行插桩 **linself**，如下所示：

```
< b := cas(&Head, h, s); if (b) linself; >
```

完成插桩之后，我们定义 I，R 和 G，并用逻辑规则推理插桩后的代码。如图 5.15所示，不变式 I 将具体链表的节点与抽象队列联系起来。抽象队列是一个值的序列 Q，抽象的 ENQ(v) 就表示为原子指令 <Q := Q::v>，抽象的 DEQ() 如下所示，它在 Q 为空时返回 EMPTY，在 Q 非空时取下第一个元素并返回。

```
DEQ() { local v;
  < if (Q = ϵ) { v := EMPTY; }
    else { v := head(Q); Q := tail(Q); }
    return v;
  >
}
```

在 I 的定义中，谓词 lsq 要求具体链表除哨兵节点外的各节点的值恰好构成序列 Q。如前所述，Tail 指针要么指向链表的最后一个节点，要么指向倒数第二个（见谓词 tls）。此外，我们引入一个辅助变量 GH 收集那些曾经从链表中取下

$$I \stackrel{\text{def}}{=} \exists h, t, A.\ (\mathtt{Q} = A) * (\mathtt{Head} = h) * (\mathtt{Tail} = t) * \mathsf{lsq}(h, t, A) * \mathsf{garb}(h)$$

$$\mathsf{lsq}(h, t, A) \stackrel{\text{def}}{=} \exists v, A', A''.\ (v :: A = A' :: A'') \wedge \mathsf{ls}(h, A', t) * \mathsf{tls}(t, \_, A'')$$

$$\mathsf{ls}(x, A, y) \stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge \mathit{emp}) \vee (x \neq y \wedge \exists z, v, A'.\ A = v :: A' \wedge \mathsf{node}(x, v, z) * \mathsf{ls}(z, A', y))$$

$$\mathsf{ls}(x, y) \stackrel{\text{def}}{=} \exists A.\ \mathsf{ls}(x, A, y) \qquad \mathsf{garb}(h) \stackrel{\text{def}}{=} \exists g.\ (\mathtt{GH} = g) * \mathsf{ls}(g, h)$$

$$\mathsf{tls}(t, x, A) \stackrel{\text{def}}{=} \exists v, v'.\ (A = v \wedge \mathsf{node}(t, v, x) \wedge x = \mathtt{null}) \vee (A = v :: v' \wedge \mathsf{last2}(t, v, x, v'))$$

$$\mathsf{tls}(t, x) \stackrel{\text{def}}{=} \exists A.\ \mathsf{tls}(t, x, A) \qquad \mathsf{tls}(t) \stackrel{\text{def}}{=} \mathsf{tls}(t, \_)$$

$$\mathsf{last2}(t, v, x, v') \stackrel{\text{def}}{=} \mathsf{node}(t, v, x) * \mathsf{node}(x, v', \mathtt{null})$$

$$\mathsf{last2}(t, x) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_, x, \_) \qquad \mathsf{last2}(t) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_)$$

$$\mathsf{node}(x, v, y) \stackrel{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x, y) \stackrel{\text{def}}{=} \mathsf{node}(x, \_, y)$$

$$R = G \stackrel{\text{def}}{=} [\mathit{Enq} \vee \mathit{Deq} \vee \mathit{Swing}]_I$$

$$\mathit{Enq} \stackrel{\text{def}}{=} \exists v, v', A, t, x.\ ((\mathtt{Q} = A) * (\mathtt{Tail} = t) * \mathsf{node}(t, v, \mathtt{null})) \ltimes ((\mathtt{Q} = A :: v') * (\mathtt{Tail} = t) * \mathsf{last2}(t, v, x, v'))$$

$$\mathit{Deq} \stackrel{\text{def}}{=} \exists v, A, h, t, x, y.\ ((\mathtt{Q} = v :: A) * (\mathtt{Head} = h) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\mathtt{Tail} = t) \wedge h \neq t) \ltimes ((\mathtt{Q} = A) * (\mathtt{Head} = x) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\mathtt{Tail} = t))$$

$$\mathit{Swing} \stackrel{\text{def}}{=} \exists v, v', t, x.\ ((\mathtt{Tail} = t) * \mathsf{last2}(t, v, x, v')) \ltimes ((\mathtt{Tail} = x) * \mathsf{last2}(t, v, x, v'))$$

图 5.15 MS 无锁队列的不变式和依赖/保证条件

的“垃圾”节点（见谓词 garb），以精确刻画共享资源 I。辅助变量 GH 在对象初始化的时候设置为 Head，之后就不再修改。由于 deq 操作并不会修改出队节点的 next 指针，所有已出队的节点就构成了从 GH 到当前 Head 的表段。

依赖/保证条件 R 和 G 刻画两层程序对应执行的状态转换。如图 5.15所示，R 和 G 一样，均包含算法在共享资源上的所有可能的动作。动作 *Enq* 和 *Deq* 分别代表enq 方法第 8 行和deq 方法第 28 行的 LP。它们分别插入和删除节点，既操作具体链表，也更改抽象队列。注意 *Deq* 要求动作发生前 Head 与 Tail 并不指向同一节点。这样 *Deq* 动作后，Head 决不会与 Tail“交叉”（如 Head 指向链表最后一个节点而 Tail 却指向倒数第二个），从而维持了不变式 I。动作 *Swing* 在 Tail 指向链表倒数第二个节点的时候推进 Tail 指针，它不会改变抽象队列。

图 5.17和图 5.18分别展示了 enq 和 deq 实现的证明过程。证明中用到的辅助断言在图 5.16中定义。该证明并不困难，它与使用 LRG 验证算法的部分正确性的证明过程类似。但此时我们需要在断言中描述抽象对象和当前线程的抽象操作，推理插桩的辅助指令。

$$\mathsf{readTail}(t,n) \stackrel{\text{def}}{=} (\texttt{Tail} = t) * \mathsf{tls}(t,n) \lor \mathsf{readTailEnvAdv}(t,n)$$

$$\mathsf{readTailEnvAdv}(t,n) \stackrel{\text{def}}{=} \exists x.\, \mathsf{node}(t,n) * \mathsf{ls}(n,x) * (\texttt{Tail} = x) * \mathsf{tls}(x)$$

$$\mathsf{readTail}(t) \stackrel{\text{def}}{=} \mathsf{readTail}(t,\_) \qquad \mathsf{readTailEnvAdv}(t) \stackrel{\text{def}}{=} \mathsf{readTailEnvAdv}(t,\_)$$

$$\mathsf{readTailNext}(t,n) \stackrel{\text{def}}{=} \mathsf{readTail}(t,n) \lor \mathsf{readTailNextNullEnv}(t,n)$$

$$\mathsf{readTailNextNullEnv}(t,n) \stackrel{\text{def}}{=} (n = \texttt{null}) \land ((\texttt{Tail} = t) * \mathsf{last2}(t)) \lor \mathsf{readTailEnvAdv}(t))$$

$$\begin{aligned}&\mathsf{readTailNextNull}(t,n) \stackrel{\text{def}}{=}\\ &\quad ((\texttt{Tail} = t) * \mathsf{node}(t,n) \land (n = \texttt{null})) \lor \mathsf{readTailNextNullEnv}(t,n)\end{aligned}$$

$$\mathsf{readTailNextNonnull}(t,n) \stackrel{\text{def}}{=} ((\texttt{Tail} = t) * \mathsf{last2}(t,n)) \lor \mathsf{readTailEnvAdv}(t,n)$$

$$\mathsf{readHead}(h,x) \stackrel{\text{def}}{=} ((h = x) \land (\texttt{Head} = x)) \lor \mathsf{readHeadEnv}(h,x)$$

$$\mathsf{readHeadEnv}(h,n,x) \stackrel{\text{def}}{=} (h \neq x) \land \mathsf{node}(h,n) * \mathsf{ls}(n,x) * (\texttt{Head} = x)$$

$$\mathsf{readHead}(h) \stackrel{\text{def}}{=} \mathsf{readHead}(h,\_) \qquad \mathsf{readHeadEnv}(h,x) \stackrel{\text{def}}{=} \mathsf{readHeadEnv}(h,n,x)$$

$$\mathsf{readHeadTail}(h,t) \stackrel{\text{def}}{=} (\exists x.\, \mathsf{readHead}(h,x) * \mathsf{ls}(x,t) * \mathsf{readTail}(t)) \lor \mathsf{readHeadTailEnv}(h,t)$$

$$\begin{aligned}&\mathsf{readHeadTailEnv}(h,t) \stackrel{\text{def}}{=}\\ &\quad \exists x,y,z.\, \mathsf{ls}(h,t) * \mathsf{node}(t,x) * \mathsf{ls}(x,y) * (\texttt{Head} = y) * \mathsf{ls}(y,z) * (\texttt{Tail} = z) * \mathsf{tls}(z)\end{aligned}$$

$$\begin{aligned}&\mathsf{readHeadNextAfterTail}(h,n,t) \stackrel{\text{def}}{=}\\ &\quad (\texttt{Head} = h) * (((h = t) \land \mathsf{readTailNext}(t,n)) \lor (\mathsf{node}(h,n) * \mathsf{ls}(n,t) * \mathsf{readTail}(t)))\\ &\quad \lor (\exists x.\, \mathsf{readHeadEnv}(h,n,x) * \mathsf{ls}(x,t) * \mathsf{readTail}(t)) \lor \mathsf{readHeadNextEnv}(h,n,t)\end{aligned}$$

$$\begin{aligned}&\mathsf{readHeadNextEnv}(h,n,t) \stackrel{\text{def}}{=}\\ &\quad \exists x,y,z.\, (((h = t) \land \mathsf{node}(t,x) \land ((x = n) \lor (n = \texttt{null}))\\ &\quad \lor (\mathsf{node}(h,n) * \mathsf{ls}(n,t) * \mathsf{node}(t,x))) * \mathsf{ls}(x,y) * (\texttt{Head} = y) * \mathsf{ls}(y,z) * (\texttt{Tail} = z)\end{aligned}$$

$$\begin{aligned}&\mathsf{readHeadNextVal}(h,n,v) \stackrel{\text{def}}{=}\\ &\quad ((\texttt{Head} = h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,\_) * (\texttt{Tail} = n))\\ &\quad \lor (\exists x,t.\, (\texttt{Head} = h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,x) * \mathsf{ls}(x,t) * (\texttt{Tail} = t))\\ &\quad \lor (\exists x,t.\, \mathsf{readHeadEnv}(h,n,x) * \mathsf{ls}(x,t) * (\texttt{Tail} = t))\end{aligned}$$

图 5.16 MS 无锁队列的证明的辅助定义

```
enq(v) {
 local x, t, s, b;
 {I * (cid ↣ (ENQ, v))}
 b := false;  x := cons(v, null);
 {((¬b) * I * toEnq) ∨ (b * I * (cid ↣ end))}
 while (!b) {
   {(¬b) * I * toEnq}
   < t := Tail; >
   {(¬b) * (I ∧ readTail(t) * true) * toEnq}
   s := t.next;
   {(¬b) * (I ∧ readTailNext(t, s) * true) * toEnq}
   if (t = Tail) {
     {(¬b) * (I ∧ readTailNext(t, s) * true) * toEnq}
     if (s = null) {
       {(¬b) * (I ∧ readTailNextNull(t, s) * true) * toEnq}
       < b := cas(&(t.next), s, x); if (b) linself; >
       { (b * (I ∧ readTailNextNonnull(t, x) * true) * (cid ↣ end))
       ∨ ((¬b) * (I ∧ readTailNextNullEnv(t, s) * true) * toEnq) }
       if (b) {
         {b * (I ∧ readTailNextNonnull(t, x) * true) * (cid ↣ end)}
         cas(&Tail, t, x);
         {b * I * (cid ↣ end)}
       }
       {(b * I * (cid ↣ end)) ∨ ((¬b) * I * toEnq)}
     } else {
       {(¬b) * (I ∧ readTailNextNonnull(t, s) * true) * toEnq}
       cas(&Tail, t, s);
       {(¬b) * I * toEnq}
     }
   }
 }
 {I * (cid ↣ end)}
}
```

Here $\mathsf{toEnq} \stackrel{\text{def}}{=} \mathsf{node}(\mathtt{x}, \mathtt{v}, \mathtt{null}) * (\mathtt{cid} \rightarrowtail (\mathtt{ENQ}, \mathtt{v}))$.

图 5.17 MS 无锁队列 `enq` 方法实现的证明

```
1  deq() {
2   local v, s, h, t, b;
    {I * (cid ↣ DEQ)}
3   b := false;
    {((¬b) * I * (cid ↣ DEQ)) ∨ (b * I * (cid ↣ (end, v)))}
4   while (!b) {
      {(¬b) * I * (cid ↣ DEQ)}
5     < h := Head; >
      {(¬b) * (I ∧ readHead(h) * true) * (cid ↣ DEQ)}
6     < t := Tail; >
      {(¬b) * (I ∧ readHeadTail(h, t) * true) * (cid ↣ DEQ)}
7     < s := h.next; if (h = t && s = null) trylinself; >
      { (¬b) * (I ∧ readHeadNextAfterTail(h, s, t) * true)
        * ((h = t ∧ s = null ∧ (cid ↣ DEQ ⊕ cid ↣ (end, EMPTY)))
          ∨ ((h ≠ t ∨ s ≠ null) ∧ (cid ↣ DEQ))) }
8     if (h = Head) {
9       if (h = t) {
10        if (s = null) {
            {(¬b) * I * (h = t ∧ s = null ∧ (cid ↣ DEQ ⊕ cid ↣ (end, EMPTY)))}
11          commit(cid ↣ (end, EMPTY));
            {(¬b) * I * (cid ↣ (end, EMPTY))}
12          v := EMPTY;  b := true;
            {b * I * (cid ↣ (end, v))}
13        } else {
            {(¬b) * (I ∧ readTailNextNonnull(t, s) * true) * (cid ↣ DEQ)}
14          cas(&Tail, t, s);
            {(¬b) * I * (cid ↣ DEQ)}
15        }
16      } else {
          {(¬b) * (I ∧ readHeadNextAfterTail(h, s, t) * true) * (cid ↣ DEQ) ∧ (h ≠ t)}
17        v := s.val;
          {(¬b) * (I ∧ readHeadNextVal(h, s, v) * true) * (cid ↣ DEQ)}
18        < b := cas(&Head, h, s); if (b) linself; >
          {((¬b) * I * (cid ↣ DEQ)) ∨ (b * I * (cid ↣ (end, v)))}
19      }
20    } else {
        {(¬b) * I * ((cid ↣ DEQ ⊕ cid ↣ (end, EMPTY)) ∨ (cid ↣ DEQ))}
21      commit(cid ↣ DEQ);
        {(¬b) * I * (cid ↣ DEQ)}
22    }
23  }
    {I * (cid ↣ (end, v))}
24  return v;
    {I * (cid ↣ (end, v))}
25 }
```

图 5.18 MS 无锁队列 deq 方法实现的证明

```
 1 CCAS(o, n) {                  12 Complete(d) {
 2   local r, d;                 13   local b;
 3   d := cons(cid, o, n);       14   b := flag;
 4   r := cas1(&a, o, d);        15   if (b)
 5   while(IsDesc(r)) {          16     cas1(&a, d, d.n);
 6     Complete(r);              17   else
 7     r := cas1(&a, o, d);      18     cas1(&a, d, d.o);
 8   }                           19 }
 9   if(r = o) Complete(d);      20 SetFlag(b) {
10   return r;                   21   flag := b;
11 }                             22 }
```

图 5.19 CCAS 代码

5.5.3　CCAS 算法

CCAS（conditional compare-and-swap）算法 [17] 是 RDCSS 算法 [46] 的简化版本。它同时涉及了帮助机制和依赖未来的 LP。图 5.19展示了它的代码。

该并发对象包含一个整型变量 a 和一个布尔变量 flag。方法SetFlag（见图 5.19第 20 行）可以直接修改 flag。方法CCAS 需要两个参数：o 是 a 的预期的旧值，n 是 a 的新值。当 flag 为 **true** 而且预期值 o 与 a 相同时，将 a 更新为 n；否则什么都不做。CCAS 永远返回 a 的（实际）旧值。

图 5.19中的实现用到cas 指令的一个变种，我们记作cas1。不同于cas 返回一个指示是否成功的布尔值，cas1(&a,o,n) 返回存储在 a 中的旧值。方法CCAS 开始的时候，线程首先创建自己的描述符（第 3 行），它包含线程标识号和调用 CCAS 的参数。然后线程用cas1 指令尝试将自己的描述符放在 a 里（第 4 行）。如果成功，说明 a 的实际旧值 r 等于预期旧值 o （即第 9 行的判断成功），线程就调用辅助函数Complete。根据 flag 是不是 **true**，Complete 选择将 a 置为新值 n（第 16 行）或是将 a 恢复为旧值 o （第 18 行）。而如果第 4 行的cas1 指令发现 a 包含的是某个线程描述符（即IsDesc 成立），当前线程就会在做自己的操作之前先尝试帮助完成描述符中的操作（第 6 行）。这里我们为了简化程序语言而不允许嵌套函数调用，因此应把辅助函数Complete 看作一个宏定义。

该算法的 LP 在第 4 行，第 7 行和第 14 行。在第 4 行和第 7 行的时候，如果发现 a 含有一个不同于 o 的值（不是线程描述符），则整个 CCAS 操作失败，这两行就是当前线程的 LP。因此我们如下插桩这两行：

```
<r := cas1(&a, o, d); if(r!=o && !IsDesc(r)) linself;>
```

如果当前线程成功地把描述符 d 置于 a 内，则它的 LP 应在Complete 函数中。注意任何线程都可能调用Complete 来帮助完成操作，因此 LP 应在某个执行

$$I \stackrel{\text{def}}{=} (\texttt{flag} = \texttt{flag}_a) * (\mathsf{aVal} \vee \mathsf{aDesc}) * \mathsf{garb}$$

$$\mathsf{aVal} \stackrel{\text{def}}{=} (\texttt{a} = \texttt{a}_a) \wedge \neg\texttt{IsDesc}(a) \qquad \mathsf{aDesc} \stackrel{\text{def}}{=} \exists d, t, o, n.\ \mathsf{aDesc}(d, t, o, n)$$

$$\begin{aligned}\mathsf{aDesc}(d, t, o, n) \stackrel{\text{def}}{=}\ & (\texttt{a} = d) * d \mapsto (t, o, n) \\ & * (\mathsf{notDone}(t, o, n) \vee \mathsf{trySucc}(t, o, n) \vee \mathsf{tryFail}(t, o, n) \vee \mathsf{tryBoth}(t, o, n))\end{aligned}$$

$$\mathsf{notDone}(t, o, n) \stackrel{\text{def}}{=} t \rightarrowtail (\texttt{CCAS}, o, n) * (\texttt{a}_a = o)$$

$$\mathsf{endSucc}(t, o, n) \stackrel{\text{def}}{=} t \rightarrowtail (\mathbf{end}, o) * (\texttt{a}_a = n) \qquad \mathsf{endFail}(t, o) \stackrel{\text{def}}{=} t \rightarrowtail (\mathbf{end}, o) * (\texttt{a}_a = o)$$

$$\mathsf{trySucc}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endSucc}(t, o, n)$$

$$\mathsf{tryFail}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endFail}(t, o)$$

$$\mathsf{tryBoth}(t, o, n) \stackrel{\text{def}}{=} \mathsf{notDone}(t, o, n) \oplus \mathsf{endSucc}(t, o, n) \oplus \mathsf{endFail}(t, o)$$

图 5.20 CCAS 算法的不变式

第 16 行或第 18 行会成功的线程执行第 14 行的时候。它是一个依赖未来且可能在其他线程代码中的 LP。因此，我们用 **trylin**(d.id) 指令插桩第 14 行，投机地执行 d 中线程（可能不是当前线程）的抽象操作，如下所示：

```
< b := flag; if (a = d) trylin(d.id); >
```

这里条件a=d 要求描述符中的抽象操作尚未完成。之后，在第 16 行和第 18 行，我们提交正确的投机情况。下面是我们对第 16 行的插桩（其中 s 是一个局部变量）。

```
< s := cas1(&a, d, d.n);
  if(s = d) commit(d.id ↣(end, d.o) * (a_a = d.n)); >
```

它要求该cas1 指令成功的时候，一定存在下述投机情况：d 中线程已经完成抽象操作，且当前抽象对象的 a（记为 $\texttt{a}_a$，以与具体对象的 a 相区分）的值等于新值 n。类似地，第 18 行的插桩如下：

```
< s := cas1(&a, d, d.o);
  if(s = d) commit(d.id ↣(end, d.o) * (a_a = d.o)); >
```

然后我们定义 I，R 和 G，并应用推理规则证明插桩后的代码。如图 5.20所示，不变式 I 刻画了具体对象和抽象对象的 flag 和 a。为了方便定义，我们令抽象对象的 flag 和 a 带有下标 a。不变式 I 要求 flag 在两层总是相等，当 a 是普通值的时候在两层也是相等的（谓词 aVal）。当具体的 a 是某个描述符 d 时（谓词 aDesc），d 中线程 t 的抽象 CCAS 操作要么尚未完成（谓词 notDone），要么已经投机完成。若已经投机完成，则可能包含不同的投机情况，分别由谓词 trySucc，tryFail 和 tryBoth 刻画。此外，谓词 garb 记录了无用的线程描述符

和抽象操作，这些抽象操作虽然已经完成，但由于算法并不会回收线程描述符，它们仍然属于共享的资源。garb 的具体定义在此省略，可见技术报告 [53]。

依赖/保证条件 R 和 G 包括了共享资源上的所有动作。例如，图 5.19 第 4 行和第 7 行的cas1 成功时的动作定义为 *PlaceD*：

$$\begin{aligned}\mathit{PlaceD}_{\mathsf{t}} \stackrel{\text{def}}{=}\ & \exists v, d, o, n.\ ((\mathtt{a} = v) \wedge \neg \mathtt{IsDesc}(v)) \\ & \ltimes ((\mathtt{a} = d) * d \mapsto (\mathsf{t}, o, n) * \mathsf{t} \rightarrowtail (\mathtt{CCAS}, o, n))\end{aligned}$$

它将当前线程 t 的描述符 d 和抽象操作从线程的局部资源转移到共享资源中。这样，t 的抽象操作就在 pending 线程池中了，其他线程就可以帮助 t 完成。

第 14 行（插桩后）的动作保证 *TrylinSucc* $\vee$ *TrylinFail*，它同时体现了帮助和投机机制。

$$\begin{aligned}\mathit{TrylinSucc} &\stackrel{\text{def}}{=} \mathtt{flag} * (\exists t, o, n.\ \mathsf{notDone}(t, o, n) \propto \mathsf{endSucc}(t, o, n)) \\ \mathit{TrylinFail} &\stackrel{\text{def}}{=} (\neg \mathtt{flag}) * (\exists t, o, n.\ \mathsf{notDone}(t, o, n) \propto \mathsf{endFail}(t, o))\end{aligned}$$

这里我们用 $p \propto q$（在图 5.8中定义）描述 **trylin** 的动作。例如 *TrylinSucc* 要求动作执行前必须有投机情况 notDone（允许同时有其他的投机情况），执行动作后增加新的投机情况 endSucc，而原有的所有投机情况都被保留。注意 *TrylinSucc* 和 *TrylinFail* 允许当前线程帮助其他某个线程 t 投机执行其抽象操作。

第 16 行和第 18 行在cas1 成功时的动作分别是 *RmvDSucc* 和 *RmvDFail*，它们会重置 a（移除 a 中的线程描述符），同时提交合适的投机情况。例如，第 16 行的动作 *RmvDSucc* 如下定义：

$$\begin{aligned}&\mathit{RmvDSucc} \stackrel{\text{def}}{=} \\ &\quad \exists d, t, o, n.\ ((\mathtt{a} = d) * d \mapsto (t, o, n) * (\mathsf{endSucc}(t, o, n) \oplus \mathsf{true}) * \mathsf{garb}) \\ &\quad \ltimes ((\mathtt{a} = n) * d \mapsto (t, o, n) * \mathsf{endSucc}(t, o, n) * \mathsf{garb'})\end{aligned}$$

这里我们用 garb' 表示线程 t 的描述符 d 和它完成抽象操作的结果 $t \rightarrowtail (\mathbf{end}, o)$ 都已被列入 garb 中。*RmvDFail* 的定义与 *RmvDSucc* 类似。

详细的证明过程可见技术报告 [53]。证明中需要注意的是，我们要保证当前线程不会“幻想”其他线程的帮助而完成操作。在 CCAS 代码的任何一个程序点，环境都可能执行 **trylin** 而帮助完成当前线程的操作。但是，无论环境是否帮助当前线程，第 16 行和第 18 行的 **commit** 都不能失败。这要求我们小心区分两种不同的不确定性：由投机引起的不确定性和由环境干扰引起的不确定性。前者允许我们丢弃错误的情况，但是对于后者，我们要考虑所有可能的情况。

5.6　本章小结与相关工作

本章我们提出了一种新颖的程序逻辑验证并发对象的线性一致性，特别地，它支持 LP 不固定的并发对象。该逻辑是在 LRG [24] 的基础上，引入了专门验证线性一致性的辅助指令，并设计了相应的推理规则。断言和依赖/保证条件是在关系式的状态上解释的，它们将具体实现和对应的抽象操作联系起来。该逻辑的元理论是一个具有可组合性的模拟关系，它是 RGSim 的扩展，能够蕴涵一种上下文精化关系。而该上下文精化关系与线性一致性等价。我们的程序逻辑和模拟关系都能够推理具有帮助机制和依赖未来的 LP 的并发对象。如表 5.1所示，我们已经应用该逻辑验证了许多经典的并发对象。

对于并发对象线性一致性的验证，有大量的相关工作。然而，绝大部分已有工作仅支持 LP 固定的算法（如 [14, 16, 20]）。下面我们主要讨论验证 LP 不固定的对象方面的研究工作。

Vafeiadis 曾扩展 RGSep 来证明线性一致性 [15]，我们的程序逻辑受到了这一工作的启发。在 Vafeiadis 的工作中，他也将抽象对象和抽象原子操作视为辅助状态和代码。我们的程序逻辑与他的工作有两个重要的区别。首先，他引入预言变量（prophecy variable）来处理依赖未来的 LP，但是，迄今为止预言变量尚没有一个令人满意的语义。而我们引入 try-commit 指令，它们具有直接的语义。其次，Vafeiadis 没有定义并证明他的程序逻辑对于线性一致性的可靠性。而我们设计了一种模拟关系作为程序逻辑的元理论，解决了这一问题。正如我们在第 5.1节所提到的，定义那样一个能支持不固定 LP 的模拟关系，是我们要解决的最困难的问题之一。最近 Vafeiadis 开发了验证线性一致性的自动工具 [57]，并证明了它对于线性一致性的可靠性。但是对于 LP 不固定的算法，这一新工作只支持那些不修改对象的方法，而无法验证表 5.1中提到的许多并发对象，如 HSY 栈、CCAS 算法和 RDCSS 算法等。

Turon 等人提出了验证细粒度并发的 logical relation 技术，能够保证具体实现和抽象对象之间的上下文精化 [17]。他们也定义了一个模拟关系作为基础理论。他们的模拟关系与我们的很相似，其中用到的“spec thread pool”对应于我们提出的 pending 线程池。我们的模拟关系中的投机的想法借鉴了他们的工作，实际上该想法可以追溯至 forward-backward 模拟关系 [48]。除了这些相似点外，我们的工作提出了一个新的程序逻辑，包括一套用来插桩的辅助指令。我们提供一套语法层面上的逻辑规则，使得人们在验证程序的时候可以直接应用这些规则，而不需要像使用 logical relation 技术进行证明时那样考虑程序的语义。我们设计的辅助指令，包括支持投机的 try-commit 指令，也为证明带来了便利。另

一方面，他们的工作支持高阶（higher-order）语言、递归类型和多态。而我们更关注并发程序验证本身，用的是一个简单的一阶语言。

O'Hearn 等人证明了懒惰链表算法的一个乐观版本的线性一致性 [58]。他们的证明的关键是一个“Hindsight 引理”，它不需要知道方法实现的 LP 位置，而是为线性一致性的成立提供了非构造性的（non-constructive）证据。Hindsight 引理能够直观地解释链表算法正确的原因。然而，其他的并发算法是否也有类似的 Hindsight 引理，仍然是一个未解之谜。

Colvin 等人用 forward 模拟关系和 backward 模拟关系的组合验证了懒惰链表算法 [47]。他们的模拟关系是全局的，因为他们需要知道所有线程的程序计数器（program counter）。此外，他们的模拟关系是专为懒惰链表算法构造的，而我们的模拟关系则更为通用，可以应用于许多算法。

Derrick 等人定义的模拟关系 [59] 是单个线程上的，而且是通用的。但是，他们要求 LP 不固定的算法不能修改对象内存，因此无法支持表 5.1中提到的许多例子，如 CCAS 算法等。后来他们还提出了一种 backward 模拟关系来验证线性一致性，并证明了该验证方法的完备性（completeness） [60]。但是，该 backward 模拟关系是全局的，不支持线程模块化的（thread modular）验证，而且他们也没有设计程序逻辑。

Elmas 等人通过增量式重写（rewriting）对象实现的细粒度代码来验证线性一致性 [61]。他们的方法不需要找到 LP。他们的重写规则基于 left/right mover 技术，但不能像我们的工作那样支持霍尔风格的推理。

此外，还有大量的基于模型检测的工具（如 [62, 63]）可以检查线性一致性。例如，Vechev 等人的工具要求用户在实现代码中插桩其他线程的动作，从而检查 LP 不固定的算法的线性一致性 [63]。他们的方法不是线程模块化的。从他们的例子中可以看出，他们需要事先知道并发执行的线程的数目。此外，尽管他们的工具可以成功检测不满足线性一致性的代码，但是当遇到线性一致的代码时他们的工具常常不终止。

第 6 章　刻画并发对象进展性的上下文精化框架

在上一章中，我们将并发对象的线性一致性验证归结为证明具体实现和抽象操作之间的一种上下文精化关系。除了线性一致性外，并发对象还应满足进展性（progress）性质，例如无等待性（wait-freedom），无锁性（lock-freedom），无阻碍性（obstruction-freedom），无饥饿性（starvation-freedom）和无死锁性（deadlock-freedom）等。本章研究这些进展性性质与上下文精化之间的关系。我们提出了一个上下文精化框架，证明了对于满足线性一致性的并发对象，每种进展性性质等价于一种对终止性敏感的（termination-sensitive）上下文精化关系。

6.1　并发对象的进展性性质与我们的上下文精化框架概述

我们首先简要回顾线性一致性和与它等价的上下文精化关系，非形式地解释各种进展性性质，并介绍我们的新的等价结果。

6.1.1　线性一致性与上下文精化

如上一章所述，线性一致性描述了并发对象实现的原子的行为。它要求对象的每个操作都仿佛是在调用和返回之间的某个时刻“起作用”。直观上，它建立了对象具体实现 Π 和抽象原子操作 Π_A 之间的一种上下文精化关系 $\Pi \sqsubseteq \Pi_A$。简单来说，如果在任何上下文（即客户端程序）中将 Π_A 替换为 Π 都不会增加程序的外部可观测行为，那么 Π 是对 Π_A 的上下文精化（$\Pi \sqsubseteq \Pi_A$ 成立）。此时，外部观测者无法通过监视客户端程序的行为而判断出 Π_A 已经被替换为 Π。

当外部可观测行为是输入/输出事件的有限路径时，相应的上下文精化已经被证明与线性一致性等价（见上一章的定理 5.6或文献 [5]）。也就是说，这一基本的上下文精化关系刻画了线性一致性。但是，它不能刻画对象的进展性或终止性性质。在下面的例子中，尽管`f` 的具体实现不终止，但 $\Pi \sqsubseteq \Pi_A$ 成立（假设对象仅含有方法`f`）。

Π(`f`) : `while(true) skip;`　　Π_A(`f`) : `skip;`

C : `print(1); f(); print(1);`

例如对于客户端程序 C，当它使用 Π 时，产生的可观测行为是 $\{\epsilon, (\mathbf{out}, 1) :: \epsilon\}$；而当它使用 Π_A 时，产生的可观测行为是 $\{\epsilon, (\mathbf{out}, 1) :: \epsilon, (\mathbf{out}, 1) :: (\mathbf{out}, 1) :: \epsilon\}$。前

者确实是后者的子集。究其原因，$\Pi \sqsubseteq \Pi_A$ 在抽象层考虑的是事件路径的 *前缀闭的*（prefix-closed）集合。

6.1.2　进展性性质

图 6.1展示了计数器对象的多种可能实现，每种实现体现了本文研究的一种进展性性质。该计数器对象提供两个方法 `inc` 和 `dec`，分别增加和减少共享变量 `x`。注意这里的实现代码仅仅是为了说明进展性性质的含义，不一定具有实用价值。

简单来讲，满足*无等待性*（wait-freedom）的对象实现保证任何线程调用的任何操作都可以在有限步内完成 [64]。图 6.1(a) 展示了一个理想化的无等待的计数器实现，其中 `inc` 和 `dec` 操作都是原子执行的。它保证了无论环境线程如何干扰，每个方法调用都一定能结束，因此它满足无等待性。注意实际的无等待的计数器实现十分复杂，往往涉及数组和原子快照 [65]。

无锁性（lock-freedom）与无等待性类似，但它仅保证*存在某个*线程能够在有限步内结束操作 [64]。第 5章中提到的许多例子都是无锁的，如 Treiber 栈，HSY 栈，MS 无锁队列等。典型的无锁算法重复使用`cas` 指令尝试更新对象直到成功，图 6.1(b) 就展示了这样的一个计数器实现。它满足无锁性，因为无论 `inc` 和 `dec` 操作如何并发，总是存在成功的更新。但是，它不满足无等待性。对于客户端程序 (6.1)，

```
inc();  ||  while(true) inc();                    (6.1)
```

右边线程可能持续更新 `x`，使得左边线程的`cas` 指令总是失败而无法完成操作。

Herlihy 等人提出了*无阻碍性*（obstruction-freedom），保证任何线程最终单独执行时可以完成操作 [66]。他们举了两个双端队列（double-ended queue）作为例子。在图 6.1(c) 中，我们人为构造了一个无阻碍的计数器实现，虽然它看起来不太自然，但可以展示无阻碍性的含义。

该实现引入了一个共享变量 `i`，方法 `inc` 在将 `i` 增加到 10 后才会原子地增加 `x`，而方法 `dec` 则需要先将 `i` 减小到 0 后才能原子地减少 `x`。当一个方法隔离执行（即没有其他线程的干扰）时，它一定会终止。因此该实现满足无阻碍性。但它不满足无锁性，因为对于并发执行 `inc` 和 `dec` 的客户端程序 (6.2)，

```
inc();  ||  dec();                                (6.2)
```

可能没有一个方法会返回。如果左边线程每次对 `i` 的增加都紧跟着右边线程对 `i` 的减小，两个线程就会都陷在循环中而无法终止。

```
1 inc() { x := x + 1; }
2 dec() { x := x - 1; }
```

(a) 无等待的（理想）实现

```
1 inc() {
2  local t, b;
3  do {
4    t := x;
5    b := cas(&x,t,t+1);
6  } while(!b);
7 }
```

(b) 无锁的实现

```
1 inc() {
2   while (i < 10) {
3     i := i + 1;
4   }
5   x := x + 1;
6 }
7 dec() {
8   while (i > 0) {
9     i := i - 1;
10  }
11  x := x - 1;
12 }
```

(c) 无阻碍的实现

```
1 inc() {
2   TestAndSet_lock();
3   x := x + 1;
4   TestAndSet_unlock();
5 }
```

(d) 无死锁的实现

```
1 inc() {
2   Bakery_lock();
3   x := x + 1;
4   Bakery_unlock();
5 }
```

(e) 无饥饿的实现

图 6.1 含有方法 `inc` 和 `dec` 的计数器对象

无等待性、无锁性和无阻碍性都是针对非阻塞（non-blocking）实现的进展性性质。对于非阻塞实现，延迟调度一个线程是不会阻止其他线程完成操作的。而对于用锁实现的对象，延迟调度一个持有锁的线程就会阻塞那些请求锁的线程。对于这类实现，它们的进展性性质有*无死锁性*（deadlock-freedom）和*无饥饿性*（starvation-freedom）。

通常无死锁性和无饥饿性是基于锁和临界区定义的。无死锁性保证总有某个线程能够成功获得锁，无饥饿性则保证每个请求锁的线程最终都能获得锁 [3]。例如，测试 -设置（test-and-set）自旋锁（spin lock） [3] 是无死锁的，但不是无饥饿的。当多个线程并发请求这种锁时，总有一个线程能成功设置锁位，但可能会有某个线程总是设置失败而一直无法获得锁。Lamport 的 bakery 锁算法 [67] 是无饥饿的，它保证线程按照请求锁的顺序获得锁。

然而，正如 Herlihy 和 Shavit 所指出的 [68]，上述基于锁的定义不够好，原因是对于并发对象，往往很难找到其中充当锁的域。因此，Herlihy 和 Shavit 建议基于方法调用来定义无死锁性和无饥饿性。他们还注意到，上述基于锁的定义隐含假设每个获得锁的线程都会最终释放锁。该假设要求*公平调度*（fair scheduling），即每个线程最终都会被调度执行。

遵照 Herlihy 和 Shavit 的建议 [68]，我们将对象的*无死锁性*定义为要求在任何*公平*的执行中，总是存在某个方法调用可以完成。在图 6.1(d) 的例子中，我们

	无等待性	无锁性	无阻碍性	无死锁性	无饥饿性
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if Isolating	Div. if Fair	(t, Div.) if Fair

表 6.1 刻画进展性性质的上下文精化关系 $\Pi \sqsubseteq \Pi_A$

用一个测试 -设置锁来同步对计数器的修改。测试 -设置锁保证了总有某个线程能够获得锁，我们知道那个线程的方法调用一定能够完成，因此该实现保证无死锁性。类似地，满足无饥饿性的对象保证公平的执行中每个方法调用都能完成。图 6.1(e) 中的计数器借助 Lamport 的 bakery 锁实现。它满足无饥饿性。bakery 锁保证了每个线程都可以获得锁，所以每个方法调用最终都可以完成。

6.1.3　我们的上下文精化框架

前面介绍的五种进展性性质都没有描述它们对客户端程序行为的影响。本章我们会定义若干个上下文精化关系，刻画使用满足每种进展性性质的对象给客户端程序行为带来的影响。我们证明了线性一致性和每种进展性性质组合起来就等价于一种对终止性敏感的上下文精化关系。表 6.1总结了我们的证明结果。

在新的上下文精化关系 $\Pi \sqsubseteq \Pi_A$ 中，我们仍将输入/输出事件视为可观测行为。但是对于每种进展性性质，我们还需观测客户端程序在分别使用具体的 Π 和抽象的 Π_A 时，在特定调度下的特定发散性（divergence）行为。这里“发散性”可大致理解为终止性的反义词。

- 对于无等待性，我们需要观测具体层和抽象层每个独立线程 t 的执行是否发散——在表 6.1中记为“(t, Div.)”。我们证明了，如果在使用满足线性一致性和无等待性的对象 Π 时客户端程序的线程 t 发散，那么当该客户端程序使用 Π_A 时其线程 t 仍然发散。

- 无锁性的情况类似，但此时我们不再考虑单个线程的发散性，而是整个客户端程序的发散性——在表 6.1中记为“Div.”。如果客户端程序在使用线性一致且无锁的对象 Π 时发散，那么当它使用 Π_A 时也会发散。

- 对于无阻碍性，我们在具体层考虑隔离执行时整个程序的发散性——在表 6.1中记为“Div. if Isolating”。在隔离的执行中，最终只有一个线程在运行。我们证明了，如果使用线性一致且无阻碍的对象 Π 的客户端程序在隔离执行中发散，那么它也会在使用 Π_A 的某个执行中发散。

- 对于无死锁性，我们在具体层只关心公平的执行——在表 6.1中记为“Div. if Fair”。
- 对于无饥饿性，我们在具体层和抽象层观测单个线程的发散性，且在具体层只关心公平的执行——在表 6.1中记为“(t, Div.) if Fair”。任何使用 Π 的线程若在公平执行中发散，则它一定会在使用 Π_A 的某个执行中发散。

我们将在第 6.3节形式化定义这些新的上下文精化关系。它们构成了一个刻画线性一致性和进展性性质的统一框架。该上下文精化框架可以作为并发对象的完整正确性的新标准。每种上下文精化关系精确地刻画了抽象操作的哪些性质被具体实现所保持。由于它被线性一致性和对应的进展性性质所蕴涵，当验证客户端程序的相关性质时，我们就可以放心地将对象的具体实现替换为抽象操作。另一方面，由于上下文精化也蕴涵线性一致性和对应的进展性性质，我们就有可能扩展已有的验证上下文精化的方法（如前面介绍的模拟关系 RGSim 等）来同时验证线性一致性和进展性性质。

6.2　形式化进展性性质

如图 6.2所示，进展性性质定义在事件路径 T 和对象实现 Π 上。不同于图 5.4定义的有限事件路径，这里的事件路径 T 也可以是由事件组成的无限序列，如下所示，它是余归纳定义的。为了讨论进展性性质，我们扩展了图 5.4中事件 e 的定义。我们增加了命令 **end**，它能够生成终止事件 $(\mathsf{t}, \textbf{term})$。命令 **end** 是一个特殊记号，程序员不能直接使用它。我们还引入了事件 $(\textbf{spawn}, n)$，它表示创建了 n 个线程。新引入的事件 $(\mathsf{t}, \textbf{term})$ 和 $(\textbf{spawn}, n)$ 都不是外部可观测事件。其他的基本设定（包括程序语义和线性一致性的定义等）与第 5.2节相同。

$$
\begin{array}{rrcl}
(\mathit{Stmt}) & C & ::= & \ldots \mid \textbf{end} \\
(\mathit{Evt}) & e & ::= & \ldots \mid (\mathsf{t}, \textbf{term}) \mid (\textbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \qquad \text{(co-inductive)}
\end{array}
$$

我们仍然沿用上一章的记号。例如 $\mathsf{tid}(e)$ 仍用于取得 e 中的线程标识号。谓词 $\mathsf{is\_inv}(e)$，$\mathsf{is\_ret}(e)$ 和 $\mathsf{is\_abt}(e)$ 仍分别表示 e 是调用事件、返回事件和出错事件。$\mathsf{match}(e_1, e_2)$ 表示调用事件与响应事件（即返回事件或对象出错事件）匹配——它们是同一线程的事件。$T(i)$ 仍表示 T 的第 i 个事件。我们用 $\mathsf{last}(T)$ 表示有限的 T 的最后一个事件。$T(1..i)$ 是由 $T(1), \ldots, T(i)$ 构成的子路径。$|T|$ 仍然表示 T 的长度；当 T 无限时，我们令 $|T| = \omega$。我们仍用 $T|_{\mathsf{t}}$ 表示 T 中所有线程标识号为 t 的事件构成的子路径。

定义 6.1. 对象实现 Π 在精化映射 φ 下满足 P，表示为 $P_\varphi(\Pi)$，当且仅当

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \implies P(T)\,.$$

$$\begin{aligned}
&\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!] \stackrel{\text{def}}{=} \\
&\quad \{(\textbf{spawn}, |W|) :: T \mid (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \cdot \\
&\qquad \vee (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{*} (\textbf{skip}, _) \vee (\lfloor W \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{*} \textbf{abort}\} \\
&\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor \stackrel{\text{def}}{=} \textbf{let}\ \Pi\ \textbf{in}\ (C_1; \textbf{end}) \parallel \ldots \parallel (C_n; \textbf{end}) \\
&|\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n| \stackrel{\text{def}}{=} n \qquad\qquad \mathsf{tnum}((\textbf{spawn}, n) :: T) \stackrel{\text{def}}{=} n
\end{aligned}$$

$$\begin{aligned}
&\mathsf{pend_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is_inv}(e) \wedge \neg\exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\} \\
&\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j)) \\
&\mathsf{prog\text{-}s}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is_ret}(T(j)) \\
&\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is_abt}(T(i)) \\
&\mathsf{sched}(T) \ \text{ iff} \\
&\quad |T| = \omega \wedge \mathsf{pend_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega \\
&\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term}) \\
&\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wait\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{starvation\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \\
&\mathsf{lock\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \qquad \mathsf{deadlock\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \\
&\mathsf{obstruction\text{-}free} \ \text{ iff } \ \mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}
\end{aligned}$$

图 6.2 形式化进展性性质

对象实现 Π 具有进展性性质 P 当且仅当它的所有事件路径具有该性质（定义 6.1）。这里我们用 $\mathcal{T}_\omega$ 生成对象的完整的事件路径。其定义（在图 6.2中）与图 5.6中定义的 $\mathcal{T}[\![W, (\sigma_c, \sigma_o)]\!]$ 相似，但 $\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!]$ 包含了完整的执行过程生成的所有有限或无限的事件路径。我们用 $(W, \mathcal{S}) \xmapsto{T}{}^{\omega} \cdot$ 表示生成 T 的无限步的执行。$\lfloor W \rfloor$ 将 **end** 添加在每个线程的末尾，这样线程执行结束时就能产生一个显式的终止事件 $(\mathsf{t}, \textbf{term})$。在每条事件路径的开始，我们插入事件 $(\textbf{spawn}, n)$，其中 n 是 W 中线程的数目。我们假设线程标识号从 1 开始连续计数，因此 n 也是 W 用到的最大线程标识号。之后我们可以用 $\mathsf{tnum}(T)$ 得到这个 n，它将用于定义公平调度。

在定义事件路径上的进展性性质之前，我们首先定义一些辅助性质。如图 6.2所示，我们用 $\mathsf{pend\_inv}(T)$ 得到 T 中所有 pending 的调用事件，这些调用事件之后没有与之匹配的响应事件。$\mathsf{prog\text{-}t}(T)$ 保证 T 里的每个方法调用都能返

$$\text{lock-free} \iff \text{wait-free} \vee \text{prog-s} \qquad \text{starvation-free} \iff \text{wait-free} \vee \neg\text{fair}$$
$$\text{obstruction-free} \iff \text{lock-free} \vee \neg\text{iso} \qquad \text{deadlock-free} \iff \text{lock-free} \vee \neg\text{fair}$$

图 6.3 进展性性质之间的关系

回。具体来讲，它要求即使调用事件 e 在 T 的某个有限子路径 $T(1..i)$ 中 pending，也一定能在之后找到相匹配的响应事件 $T(j)$。prog-s(T) 则保证如果有 pending 的方法调用，则一定*存在某个方法返回*。与 prog-t 不同，prog-s 中的返回事件 $T(j)$ 不一定与 pending 的调用事件 e 相匹配。例如下面的无限事件路径 T_1 中，线程 t_2 不断产生调用事件和返回事件，但线程 t_1 的调用事件一直没有相匹配的返回事件，因此 T_1 满足 prog-s 但不满足 prog-t。

$$T_1:\ \ (\mathsf{t}_1, f, 1) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, f, 2) :: (\mathsf{t}_2, \mathbf{ret}, 2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, f, 2) :: (\mathsf{t}_2, \mathbf{ret}, 2) :: \ldots$$

此外，abt(T) 表示 T 以一个出错事件结尾。

我们还定义了三种有用的调度条件。最基本的调度条件是 sched，它要求如果无限的 T 中存在 pending 的调用事件，那么至少有一个 pending 线程被无限经常地（infinitely often）调度。事实上，有两种原因可能引起线程 t 的调用事件 pending：要么 t 根本不再被调度，要么它虽然一直被调度但它调用的方法的执行不终止。而 sched 则排除了所有 pending 线程都不再被调度的情况。例如，下面的无限路径 T_2 不满足 sched，但 T_3 满足。上面的例子 T_1 也满足 sched。

$$T_2:\ \ (\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$
$$T_3:\ \ (\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: (\mathsf{t}_2, \mathbf{obj}) :: \ldots$$

此外，对于无限的 T，公平调度 fair(T) 要求每个不终止的线程都被无限经常地调度。这里我们用 tnum(T) 获得*程序*具有的线程总数（而不是 T 中涉及的线程数），这样如果存在从未被调度的线程，fair 就不会成立。可以看出公平调度一定满足 sched。隔离调度 iso(T) 要求最终只有一个线程被调度，比如上面的例子 T_2 和 T_3 都满足 iso。

在图 6.2的底部，我们形式化地定义事件路径上的进展性性质（公式中的参数 T 被省略）。事件路径 T 满足无等待性，即 wait-free(T) 成立，当且仅当它在 sched 调度下保证 prog-t（除非它以出错事件结尾）。无锁性 lock-free(T) 的定义类似，但它仅保证 prog-s。无饥饿性 starvation-free(T) 和无死锁性 deadlock-free(T) 在 fair 调度下保证 prog-t 和 prog-s。无阻碍性 obstruction-free(T) 在 sched 且 iso 的调度下保证 prog-t。对于上面提到的几个例子，T_1 满足 lock-free，但不满

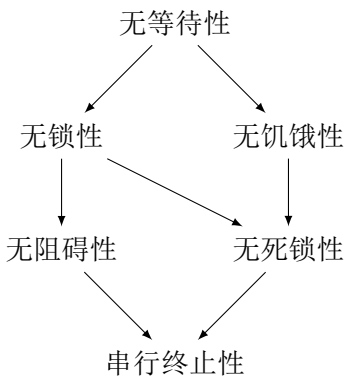


图 6.4 进展性性质的蕴涵关系格

足 wait-free。T_2 因不满足 sched，所以满足所有进展性性质。T_3 虽然满足 sched 和 iso，但不满足 prog-t，因此不满足 obstruction-free。

图 6.3中的引理展示了进展性性质之间的关系。例如，事件路径是无饥饿的，当且仅当它是无等待的或是不公平的。这些进展性性质之间的蕴涵关系构成了图 6.4中的格（lattice），其中箭头表示蕴涵。例如，无等待性蕴涵无锁性和无饥饿性，无饥饿性蕴涵无死锁性等。这个格的底元是串行环境下的进展性性质，称为“串行终止性”。

定义 6.2 (串行终止性)**.** seq-term$_\varphi(\Pi)$ 成立，当且仅当

$$\forall C_1, \sigma_c, \sigma_o, T.\, T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1), (\sigma_c, \sigma_o)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \implies \text{starvation-free}(T)\,.$$

串行终止性保证了任何单线程客户端程序的每个方法调用都能返回。我们证明了它弱于并发对象的所有进展性性质。

6.3　基于上下文精化的统一框架

我们扩展了定义 5.5中的简单上下文精化关系，使新的上下文精化等价于线性一致性和某种进展性性质。对于每种进展性性质，我们精心选择具体层和抽象层的可观测行为。

6.3.1　可观测行为

图 6.5定义了新的上下文精化用到的各种可观测行为。集合 $\mathcal{O}_\omega[\![W, (\sigma_c, \sigma_o)]\!]$ 包含了完整的执行过程中生成的所有可观测事件路径。其中 get_obsv(T) 仍然表示 T 中所有外部可观测事件（即输出事件、对象出错和客户端出错事件）构成的子路径。不同于图 5.6中定义的前缀闭集 $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$，这里我们通过 $\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!]$（见图 6.2中定义）获得完整的、可能无限的事件路径，从而能够

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (|(T|_{\mathsf{t}})| = \omega)\}$$
$$\mathcal{O}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T),\mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T),\mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{fair}(T)\}$$

图 6.5 完整事件路径的生成

观测整个程序的发散性。此外，$\mathcal{O}_{i\omega}$ 和 $\mathcal{O}_{f\omega}$ 分别得到隔离的和公平的完整执行所产生的可观测事件路径，其中 $\mathsf{iso}(T)$ 和 $\mathsf{fair}(T)$ 在图 6.2 中定义。

除了整个程序的发散性外，我们也可以观测单个线程的发散性。我们定义集合 $\mathsf{div\_tids}(T)$ 收集在事件路径 T 中发散的所有线程，然后定义 $\mathcal{O}_{t\omega}[\![W,(\sigma_c,\sigma_o)]\!]$ 获得完整执行所产生的可观测事件路径以及其中的发散线程。$\mathcal{O}_{ft\omega}[\![W,(\sigma_c,\sigma_o)]\!]$ 的定义类似，但仅考虑公平的执行。

有关发散性的更多讨论　一般来讲，发散性是终止性的反义词。例如，我们可以称程序 (6.3) 是发散的，因为它不终止。

```
x := x + 1;  ||  while(true) skip;        (6.3)
```

然而，单个线程上的发散性不等同于非终止性。一个线程若不终止，则可能有两种原因：要么它有无限的执行，要么从某一刻开始它就不再被调度。我们仅把第一种情况称作发散。例如，在一个不公平的执行中，程序 (6.3) 的左边线程可能永远不被调度，因此它没有机会终止。但它不是发散的。而对于程序 (6.4)，

```
while(true) skip;  ||  while(true) skip;        (6.4)
```

尽管有可能在某个执行中，它的单个线程不发散，但整个程序是发散的。

6.3.2　新的上下文精化关系和等价结果

表 6.2定义了对终止性敏感的上下文精化关系。每个新的上下文精化关系都遵照定义 5.5中的基本上下文精化的结构，只是采用不同的可观测行为（列举在表 6.2中）。例如，对应无等待性的上下文精化关系的完整定义如下：

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_{\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

表 6.2 进展性性质 P 对应的上下文精化关系 $\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$

$$
\begin{aligned}
&\Pi \sqsubseteq_{\varphi}^{\text{wait-free}} \Pi_A \quad \text{iff} \\
&\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \Longrightarrow \\
&\mathcal{O}_{t\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o))]\!] \subseteq \mathcal{O}_{t\omega}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a)]\!]\,.
\end{aligned}
$$

定理 6.3告诉我们，线性一致性和进展性性质 P 合起来等价于对应的上下文精化关系 $\sqsubseteq_{\varphi}^{P}$。

定理 6.3 (等价性)**.** $\Pi \preceq_{\varphi} \Pi_A \wedge P_{\varphi}(\Pi) \iff \Pi \sqsubseteq_{\varphi}^{P} \Pi_A$，其中 P 为 wait-free，lock-free，obstruction-free，deadlock-free 或 starvation-free。

这里我们假设对象的抽象操作 Π_A 绝不会阻塞，即在任何状态下调用 Π_A 中的方法都一定会返回。该定理的证明可见附录 B。

无等待性对应的上下文精化关系 $\Pi \sqsubseteq_{\varphi}^{\text{wait-free}} \Pi_A$ 在具体层和抽象层都采用 $\mathcal{O}_{t\omega}$，它将单个线程的发散性当作可观测的行为。相应的等价性可以如下理解。无等待的对象 Π 保证了每个方法调用都能完成，因此，如果某个使用 Π 的线程发散，则应怪罪于线程的客户端代码本身（如自身是个无限循环等）。换句话说，即便线程使用抽象对象 Π_A，它仍然会发散。

举例来说，考虑客户端程序 (6.1)。直观上，当该客户端程序使用抽象操作时，在任何执行中只有右边的线程 t_2 发散。因此，抽象层的 $\mathcal{O}_{t\omega}$ 是一个单元素集合 $\{(\epsilon, \{\mathsf{t}_2\})\}$。当线程使用图 6.1(a) 中的无等待的对象时，具体层的 $\mathcal{O}_{t\omega}$ 集合仍然是 $\{(\epsilon, \{\mathsf{t}_2\})\}$，它没有产生更多的可观测行为。但是如果使用一个不满足无等待性的对象（例如图 6.1(b) 中的对象），那么左边线程 t_1 不一定会终止。此时，$\mathcal{O}_{t\omega}$ 集合变为 $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$。它比抽象的客户端程序产生了更多的可观测行为，破坏了上下文精化关系。从这个例子可以看出，通过观测 div_tids 收集发散的线程，我们得以排除那些可能使更多线程发散的不满足无等待性的对象。

$\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ 采用更粗糙的可观测行为，它在具体层和抽象层使用 $\mathcal{O}_{\omega}$ 观测整个客户端程序的发散性。直观上，无锁对象 Π 保证了总是存在方法调用会完成，因此，使用 Π 的客户端程序发散就意味着它有无限多次方法调用。那么，当它使用抽象对象 Π_A 时也一定会发散。

例如考虑客户端程序 (6.1)。无论它使用图 6.1(b) 中的无锁对象，还是使用抽象对象，整个客户端程序的每次执行都发散。可观测行为集合 $\mathcal{O}_{\omega}$ 在具体层和抽象层都是 $\{\epsilon\}$。另一方面，客户端程序 (6.5) 一定会终止，且打印出 1 和 2。它

在具体层和抽象层的 $\mathcal{O}_\omega$ 集合都是 $\{1::2::\epsilon, 2::1::\epsilon\}$。

```
inc(); print(1);   ‖   dec(); print(2);
```
(6.5)

但是，如果 (6.5) 使用图 6.1(c) 中的不满足无锁性的对象，则它可能发散而不会打印出任何东西。此时 $\mathcal{O}_\omega$ 集合变为 $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$。它比抽象层产生了更多的可观测行为，因此 $\Pi \sqsubseteq_\varphi^{\text{lock-free}} \Pi_A$ 不成立。

无阻碍性是隔离执行下的进展性。相应地，$\Pi \sqsubseteq_\varphi^{\text{obstruction-free}} \Pi_A$ 也将我们的考虑限制为隔离执行。它在具体层观测 $\mathcal{O}_{i\omega}$，在抽象层采用 $\mathcal{O}_\omega$。

为了理解相应的等价性，我们考虑客户端程序 (6.5)。当它使用图 6.1(c) 中的无阻碍的对象时，在隔离执行下最终只有一个线程在运行，可以看出它不会发散，它一定能打印出 1 和 2。其具体层的 $\mathcal{O}_{i\omega}$ 集合为 $\{1::2::\epsilon, 2::1::\epsilon\}$，与抽象层的 $\mathcal{O}_\omega$ 集合一样。不满足无阻碍性的对象一般在某些隔离执行下不能保证进展性。如果客户端程序 (6.5) 使用图 6.1(d) 或图 6.1(e) 中的对象，则集合 $\mathcal{O}_{i\omega}$ 变为 $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$，不再是抽象层 $\mathcal{O}_\omega$ 的子集。多余的可观测事件路径 ϵ 由不公平的执行产生，其中一个线程获得锁然后被暂停，另一线程在隔离执行下就会持续不断地请求锁而无法终止。

$\Pi \sqsubseteq_\varphi^{\text{deadlock-free}} \Pi_A$ 在具体层使用 $\mathcal{O}_{f\omega}$，从而排除了由不公平的调度引起的期望外的发散行为。对于客户端程序 (6.5)，当它使用图 6.1(d) 或图 6.1(e) 中的对象时，集合 $\mathcal{O}_{f\omega}$ 与抽象层的 $\mathcal{O}_\omega$ 集合一样。

对于无饥饿性对应的上下文精化，我们像对无死锁性一样仍然在具体层只考虑公平执行，但也像对无等待性一样在两层观测单个线程的发散性而非整个程序的发散性。$\Pi \sqsubseteq_\varphi^{\text{starvation-free}} \Pi_A$ 在具体层使用 $\mathcal{O}_{ft\omega}$，在抽象层使用 $\mathcal{O}_{t\omega}$。对于客户端程序 (6.5)，当它使用图 6.1(e) 中的无饥饿的对象时，没有线程在公平调度下发散。也就是说，具体层的可观测行为集合 $\mathcal{O}_{ft\omega}$ 为 $\{(1::2::\epsilon, \emptyset), (2::1::\epsilon, \emptyset)\}$，它与抽象层的 $\mathcal{O}_{t\omega}$ 集合一样。

观测单个线程的发散性是我们得以区分无饥饿性和无死锁性的关键。考虑客户端程序 (6.1) 且假设公平调度。当它使用图 6.1(e) 中的无饥饿的对象时，只有右边的线程 t_2 会发散，集合 $\mathcal{O}_{ft\omega}$ 为 $\{(\epsilon, \{\mathsf{t}_2\})\}$。它与抽象层的 $\mathcal{O}_{t\omega}$ 一样。但当它使用图 6.1(d) 中的无死锁的对象时，集合 $\mathcal{O}_{ft\omega}$ 变为 $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$，破坏了上下文精化关系。

6.4　本章小结与相关工作

本章介绍了一个上下文精化框架，它将常见的五种进展性性质统一起来。对于满足线性一致性的并发对象，每种进展性性质等价于一种对终止性敏感的上下文精化关系，总结在表 6.1中。该框架让我们可以在验证客户端程序的安全性和活性性质的时候，将对象的具体实现替换为抽象的原子操作，做到模块化验证。它也使我们有希望扩展已有的验证上下文精化的方法，来同时验证线性一致性和进展性性质，我们将这项工作列为今后的研究工作之一。

尽管前人有大量的研究工作讨论这五种进展性性质，但像我们这样讨论进展性性质和上下文精化之间的联系、统一这五种进展性性质的工作并不是很多。

Gotsman 和 Yang 提出了一种新的线性一致性的定义，它可以保持无锁性；他们的工作也表明无锁性与一种对终止性敏感的上下文精化之间有一定联系 [21]。这里我们没有重新定义线性一致性，而是提出了一个统一的框架，形式化地证明了所有五种进展性性质加上线性一致性时与上下文精化之间的等价关系。

Herlihy 和 Shavit 非形式化地讨论了这五种进展性性质 [68]。我们在第 6.2节中给出的形式化定义基本遵照他们的自然语言的解释，但我们填补了程序语义与他们的基于历史的解释之间的空白。我们还注意到，他们的无阻碍性的定义对于某些例子是不妥当的（见附录 A），因此我们提出了一个不同的定义，它可能更接近通常的理解 [3]。此外，他们的工作比较了这五种进展性性质对调度的不同假设；而我们将进展性性质和上下文精化联系起来，从外延（即对客户端程序行为的影响）的角度理解进展性性质。

Fossati 等人提出了一种统一的办法在 π-演算中定义进展性性质以及他们的可观测的近似 [69] 。他们的技术设定与我们的完全不同。而且，他们对于无锁性和无等待性的可观测近似是严格弱于无锁性和无等待性的。他们没有形式化无死锁性和无饥饿性，也没有对无阻碍性给出可观测的近似。相比之下，我们对于五种进展性性质中的每一种都找到了一种等价的上下文精化关系。

还有许多工作基于时序逻辑定义进展性性质。例如，Petrank 等人 [70] 和 Dongol [71] 用线性时序逻辑分别形式化了三种非阻塞的性质和所有五种进展性性质。他们的定义使得模型检测更加容易，例如 Petrank 等人还开发了模型检测无锁性的变种的工具；而我们的上下文精化框架则可能对霍尔风格的模块化验证更有帮助。

第 7 章　结论

许多程序验证问题都可以归结为精化验证。在并发环境下，精化验证的典型应用包括：并发程序编译/优化的验证，并发对象正确性的验证，以及并发垃圾收集器的验证等。本文主要研究并发程序精化的验证技术及其应用，大致分为以下几个方面：

- 我们提出了一种基于依赖/保证的模拟关系 RGSim，作为并发程序精化的通用验证技术。
- 基于 RGSim，我们提出了一种通用的验证并发垃圾收集算法的框架。
- 我们设计了一种程序逻辑，验证并发对象的线性一致性。
- 我们提出了一个上下文精化框架，刻画并发对象的各种进展性性质。

具体来讲，我们的 RGSim 关系（定义在第 2章）将程序与并发环境之间的交互作为模拟关系的参数，以依赖/保证条件的形式表示。它具有并发可组合性，因此我们可以将多线程程序精化的验证分解为单个线程上的证明。

RGSim 将精化应用中对并发环境的特定前提参数化，因而具有较好的灵活性和实用性。例如，我们可以应用 RGSim 推理并发环境下程序优化的正确性。我们在第 3章给出了一些基于 RGSim 的关系式推理规则，刻画并证明常见优化在并发设定下的正确性。我们考察的优化包括：循环不变代码外提，强度削弱和归纳变量删除，死代码删除，冗余代码引入，等等。

在第 4章我们将并发垃圾收集（GC）算法的验证问题归约为程序变换的验证，并基于 RGSim 提出一个通用的 GC 验证框架（定理 4.1）。我们用这个框架验证了 Boehm 等人提出的并发垃圾收集算法 [23]。

此外，RGSim 还可用于验证细粒度并发程序或并发对象的正确性。它可以保证一种上下文精化关系，该上下文精化关系与并发对象的线性一致性等价。但是，RGSim 不支持可线性化点（LP）不固定的并发对象。

我们在第 5章提出了一个程序逻辑验证 LP 不固定的并发对象的线性一致性。我们的逻辑以一元并发程序逻辑 LRG [24] 为基础，但断言和依赖/保证条件都在具体状态和抽象状态的关系上解释。我们还引入了专门验证线性一致性的辅助指令，并为它们设计了新的推理规则。对于带有帮助机制的对象，我们引

入了 pending 线程池；对于依赖未来的 LP，我们引入了 try-commit 机制。我们成功地应用我们的程序逻辑验证了 12 个著名的算法（见表 5.1）。其中有些已经在 `java.util.concurrent` 包中使用，例如 MS 无锁队列 [54] 和 Harris-Michael 无锁链表 [55, 56] 等。

我们还设计了一种新的模拟关系作为该程序逻辑的元理论。它扩展了 RGSim 以支持 LP 不固定的并发对象。它仍然具有可组合性，而且能蕴涵与线性一致性等价的上下文精化关系，从而可以保证并发对象的线性一致性。我们的程序逻辑能够保证上下文精化，这本身就是一件有意义的事，因为上下文精化也许是一种更自然的、被更广泛地接受的并发对象正确性定义。

最后，在第 6章我们将并发对象的进展性性质和上下文精化联系起来。我们形式化了五种最常用的进展性性质定义：无等待性，无锁性，无阻碍性，无饥饿性，以及无死锁性。它们之间的关系构成了一个格，如图 6.4所示。我们设计了一个统一框架，用上下文精化来刻画这些进展性性质。我们证明了，对于满足线性一致性的对象，每种进展性性质与一种对程序终止性敏感的上下文精化关系等价。

我们的上下文精化框架可以视作并发对象的完整正确性的新定义。它使我们可以模块化地验证客户端程序，即在验证客户端程序时将对象的具体实现替换为抽象的原子操作。它也给了我们同时验证线性一致性和进展性性质的可能性，我们可以扩展上下文精化的验证方法（如 RGSim）来验证线性一致性和进展性。

进一步的工作

- *验证保终止性的精化*。所谓“保终止性”是指，在精化关系 $C \sqsubseteq \mathbb{C}$ 中，如果 $\mathbb{C}$ 能够终止，那么 C 也一定能终止。大部分已有的并发程序精化验证技术（包括本文中的模拟关系和程序逻辑）并不考虑保终止性，这使得不终止的 C 可以是任意程序的精化。我们希望设计一个新的模拟关系以及一个新的霍尔风格的程序逻辑，二者均能够模块化地验证保终止性的并发程序精化。

- *同时验证线性一致性和进展性性质的验证框架*。本文介绍的上下文精化框架使得我们可以用对终止性敏感的上下文精化关系来验证线性一致性和相应的进展性性质。本文还给出了一个程序逻辑，可验证终止性不敏感的上下文精化关系，进而保证线性一致性。一个自然的问题是：我们是否能扩展该程序逻辑来验证各种对终止性敏感的上下文精化关系呢？更进一步，

我们是否能有一个统一的程序逻辑（或许提供一些可实例化的参数和可替换的规则）来验证所有这些上下文精化关系呢？

- 其他应用。在精化验证的各种应用中，本文主要关注的是并发对象的验证。我们还希望探索现实世界中的其他应用，如验证软件事务内存的实现，并发程序编译器，操作系统等。

- 工具支持。将验证过程自动化也是可追求的目标之一。我们希望开发专门的工具，输入两个程序，就能够自动或半自动（允许用户适当的交互）地证明程序之间的精化关系。对于线性一致性的验证，我们希望未来可以将我们的程序逻辑在证明辅助工具 Coq 中实现，并且开发工具使验证过程自动化。

参考文献

[1] Leroy X. A Formally Verified Compiler Back-end. J. Autom. Reason., 2009, 43(4):363–446.

[2] Hoare C A R. Proof of Correctness of Data Representations. Acta Inf., 1972, 1(4):271–281.

[3] Herlihy M, Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann, April, 2008.

[4] Herlihy M P, Wing J M. Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst., 1990, 12(3):463–492.

[5] Filipović I, O'Hearn P, Rinetzky N, et al. Abstraction for concurrent objects. Theor. Comput. Sci., 2010, 411(51-52):4379–4398.

[6] Dice D, Shalev O, Shavit N. Transactional Locking II. Proceedings of DISC, Berlin, Heidelberg: Springer, 2006. 194–208.

[7] Klein G, Andronick J, Elphinstone K, et al. seL4: Formal Verification of an Operating-system Kernel. Commun. ACM, 2010, 53(6):107–115.

[8] He J, Hoare C A R, Sanders J W. Data Refinement Refined. Proceedings of ESOP. Springer, 1986. 187–196.

[9] Abadi M, Lamport L. The Existence of Refinement Mappings. Theor. Comput. Sci., 1991, 82(2):253–284.

[10] Treiber R K. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[11] Brookes S D. Full Abstraction for a Shared-Variable Parallel Language. Inf. Comput., 1996, 127(2):145–163.

[12] Hur C K, Dreyer D. A Kripke Logical Relation Between ML and Assembly. Proceedings of POPL, New York, NY, USA: ACM Press, 2011. 133–146.

[13] Abadi M, Plotkin G. A model of cooperative threads. Proceedings of POPL, New York, NY, USA: ACM Press, 2009. 29–40.

[14] Amit D, Rinetzky N, Reps T, et al. Comparison Under Abstraction for Verifying Linearizability. Proceedings of CAV, Berlin, Heidelberg: Springer, 2007. 477–490.

[15] Vafeiadis V. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July, 2008.

[16] Derrick J, Schellhorn G, Wehrheim H. Mechanically Verified Proof Obligations for Linearizability. ACM Trans. Program. Lang. Syst., 2011, 33(1):4:1–4:43.

[17] Turon A J, Thamsborg J, Ahmed A, et al. Logical Relations for Fine-grained Concurrency. Proceedings of POPL, New York, NY, USA: ACM Press, 2013. 343–356.

[18] Hendler D, Shavit N, Yerushalmi L. A Scalable Lock-free Stack Algorithm. Proceedings of SPAA, New York, NY, USA: ACM Press, 2004. 206–215.

[19] Heller S, Herlihy M, Luchangco V, et al. A Lazy Concurrent List-based Set Algorithm. Proceedings of OPODIS, Berlin, Heidelberg: Springer, 2006. 3–16.

[20] Turon A, Wand M. A separation logic for refining concurrent objects. Proceedings of POPL, New York, NY, USA: ACM Press, 2011. 247–258.

[21] Gotsman A, Yang H. Liveness-preserving atomicity abstraction. Proceedings of ICALP. Springer, 2011. 453–465.

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

[23] Boehm H J, Demers A J, Shenker S. Mostly Parallel Garbage Collection. Proceedings of PLDI, New York, NY, USA: ACM Press, 1991. 157–164.

[24] Feng X. Local rely-guarantee reasoning. Proceedings of POPL, New York, NY, USA: ACM Press, 2009. 315–327.

[25] Boehm H J. Threads cannot be implemented as a library. Proceedings of PLDI, New York, NY, USA: ACM Press, 2005. 261–268.

[26] Boehm H J, Adve S V. Foundations of the C++ concurrency memory model. Proceedings of PLDI, New York, NY, USA: ACM Press, 2008. 68–78.

[27] Benton N. Simple relational correctness proofs for static analyses and program transformations. Proceedings of POPL, New York, NY, USA: ACM Press, 2004. 14–25.

[28] Ševčík J, Vafeiadis V, Nardelli F Z, et al. Relaxed-Memory Concurrency and Verified Compilation. Proceedings of POPL, New York, NY, USA: ACM Press, 2011. 43–54.

[29] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.3, 2010. http://coq.inria.fr.

[30] Liang H, Feng X, Fu M. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. Technical report (with Coq implementation), University of Science and Technology of China, October, 2011. http://kyhcs.ustcsz.edu.cn/relconcur/rgsim.

[31] Reynolds J C. Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of LICS, Washington, DC, USA: IEEE Computer Society, 2002. 55–74.

[32] Turon A, Dreyer D, Birkedal L. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency. Proceedings of ICFP'13, New York, NY, USA: ACM Press, 2013. 377–390.

[33] Yang H. Relational separation logic. Theor. Comput. Sci., 2007, 375(1-3):308–334.

[34] Wand M. Compiler Correctness for Parallel Languages. Proceedings of FPCA, New York, NY, USA: ACM Press, 1995. 120–134.

[35] Gladstein D S, Wand M. Compiler Correctness for Concurrent Languages. Proceedings of COORDINATION, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996. 231–248.

[36] Lochbihler A. Verifying a Compiler for Java Threads. Proceedings of ESOP, Berlin, Heidelberg: Springer, 2010. 427–447.

[37] Burckhardt S, Musuvathi M, Singh V. Verifying Local Transformations on Relaxed Memory Models. Proceedings of CC/ETAPS, Berlin, Heidelberg: Springer, 2010. 104–123.

[38] Barabash K, Ben-Yitzhak O, Goft I, et al. A parallel, incremental, mostly concurrent garbage collector for servers. ACM Trans. Program. Lang. Syst., 2005, 27(6):1097–1146.

[39] Parkinson M, Bornat R, Calcagno C. Variables as Resource in Hoare Logics. Proceedings of LICS, Washington, DC, USA: IEEE Computer Society, 2006. 137–146.

[40] Vechev M T, Yahav E, Bacon D F. Correctness-Preserving Derivation of Concurrent Garbage Collection Algorithms. Proceedings of PLDI, New York, NY, USA: ACM Press, 2006. 341–353.

[41] Pavlovic D, Pepper P, Smith D R. Formal derivation of concurrent garbage collectors. Proceedings of MPC, 2010. 353–376.

[42] Kapoor K, Lodaya K, Reddy U. Fine-grained concurrency with Separation Logic. J. Philosophical Logic, 2011, 40(5):583–632.

[43] McCreight A, Shao Z, Lin C, et al. A General Framework for Certifying Garbage Collectors and Their Mutators. Proceedings of PLDI, New York, NY, USA: ACM Press, 2007. 468–479.

[44] O'Hearn P W. Resources, Concurrency, and Local Reasoning. Theor. Comput. Sci., 2007, 375(1-3):271–307.

[45] Qadeer S, Sezgin A, Tasiran S. Back and Forth: Prophecy Variables for Static Verification of Concurrent Programs. Technical Report MSR-TR-2009-142, Microsoft, October, 2009.

[46] Harris T L, Fraser K, Pratt I A. A Practical Multi-word Compare-and-Swap Operation. Proceedings of DISC, London, UK, UK: Springer, 2002. 265–279.

[47] Colvin R, Groves L, Luchangco V, et al. Formal verification of a lazy concurrent list-based set algorithm. Proceedings of CAV. Springer, 2006. 475–488.

[48] Lynch N A, Vaandrager F W. Forward and Backward Simulations: I. Untimed Systems. Inf. Comput., 1995, 121(2):214–233.

[49] Doherty S, Groves L, Luchangco V, et al. Formal Verification of a Practical Lock-Free Queue Algorithm. Proceedings of FORTE. Springer, 2004. 97–114.

[50] Gotsman A, Yang H. Linearizability with Ownership Transfer. Proceedings of CONCUR. Springer, 2012. 256–271.

[51] O'Hearn P W, Yang H, Reynolds J C. Separation and Information Hiding. Proceedings of POPL, New York, NY, USA: ACM Press, 2004. 268–280.

[52] Vafeiadis V. Concurrent Separation Logic and Operational Semantics. Proceedings of MFPS. Elsevier Science Publishers Ltd., 2011. 335–351.

[53] Liang H, Feng X. Modular Verification of Linearizability with Non-Fixed Linearization Points. Technical report, University of Science and Technology of China, March, 2013. http://kyhcs.ustcsz.edu.cn/relconcur/lin.

[54] Michael M M, Scott M L. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. Proceedings of PODC, New York, NY, USA: ACM Press, 1996. 267–275.

[55] Harris T L. A Pragmatic Implementation of Non-blocking Linked-Lists. Proceedings of DISC, London, UK, UK: Springer, 2001. 300–314.

[56] Michael M M. High Performance Dynamic Lock-free Hash Tables and List-based Sets. Proceedings of SPAA, New York, NY, USA: ACM Press, 2002. 73–82.

[57] Vafeiadis V. Automatically Proving Linearizability. Proceedings of CAV. Springer, 2010. 450–464.

[58] O'Hearn P W, Rinetzky N, Vechev M T, et al. Verifying Linearizability with Hindsight. Proceedings of PODC. ACM Press, 2010. 85–94.

[59] Derrick J, Schellhorn G, Wehrheim H. Verifying Linearisability with Potential Linearisation Points. Proceedings of FM. Springer, 2011. 323–337.

[60] Schellhorn G, Wehrheim H, Derrick J. How to Prove Algorithms Linearisable. Proceedings of CAV. Springer, 2012. 243–259.

[61] Elmas T, Qadeer S, Sezgin A, et al. Simplifying Linearizability Proofs with Reduction and Abstraction. Proceedings of TACAS, Berlin, Heidelberg: Springer, 2010. 296–311.

[62] Liu Y, Chen W, Liu Y A, et al. Model Checking Linearizability via Refinement. Proceedings of FM. Springer, 2009. 321–337.

[63] Vechev M T, Yahav E, Yorsh G. Experience with Model Checking Linearizability. Proceedings of SPIN. Springer, 2009. 261–278.

[64] Herlihy M. Wait-free Synchronization. ACM Trans. Program. Lang. Syst., 1991, 13(1):124–149.

[65] Aspnes J, Herlihy M. Wait-free Data Structures in the Asynchronous PRAM Model. Proceedings of SPAA, New York, NY, USA: ACM, 1990. 340–349.

[66] Herlihy M, Luchangco V, Moir M. Obstruction-Free Synchronization: Double-Ended Queues as an Example. Proceedings of ICDCS, Washington, DC, USA: IEEE Computer Society, 2003. 522–529.

[67] Lamport L. A New Solution of Dijkstra's Concurrent Programming Problem. Commun. ACM, 1974, 17(8):453–455.

[68] Herlihy M, Shavit N. On the Nature of Progress. Proceedings of OPODIS. Springer, 2011. 313–328.

[69] Fossati L, Honda K, Yoshida N. Intensional and extensional characterisation of global progress in the π-calculus. Proceedings of CONCUR. Springer, 2012. 287–301.

[70] Petrank E, Musuvathi M, Steesngaard B. Progress Guarantee for Parallel Programs via Bounded Lock-freedom. Proceedings of PLDI, New York, NY, USA: ACM Press, 2009. 144–154.

[71] Dongol B. Formalising Progress Properties of Non-blocking Programs. Proceedings of ICFEM, 2006. 284–303.

[72] Liang H, Hoffmann J, Feng X, et al. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. Technical report, University of Science and Technology of China, April, 2013. http://kyhcs.ustcsz.edu.cn/relconcur/prog.

附录 A　与 Herlihy 和 Shavit 的无阻碍性定义的比较

Herlihy 和 Shavit 基于*均匀隔离*（uniformly isolating）执行的概念来定义无阻碍性（obstruction-freedom）[68]。所谓均匀隔离的完整事件路径，是指对于任何执行无穷多步的线程，对于任何 $k > 0$，都能够在路径中找到该线程连续（不被打断地）执行 k 步的片断。他们的无阻碍性就要求在每个均匀隔离的事件路径上保证无等待性。此外，Herlihy 和 Shavit 还提出了一种新的进展性性质，称作无冲突性（clash-freedom），要求在均匀隔离的事件路径上保证无锁性。

下面我们通过一个简单的例子说明 Herlihy 和 Shavit 的定义与通常对无阻碍性的直观理解不一致。如下所示，该对象包含三个共享变量 x，a 和 b，提供两个方法f 和g。

```
f() {                          g() {
  while (a <= x <= b) {          while (a <= x <= b) {
    x++;                           x--;
    a--;                           b++;
  }                              }
}                              }
```

我们可以看出，如果f()或g()最终隔离执行（即，我们暂停其他所有线程，仅执行当前线程）的话，就一定能返回。因此，直观上该对象满足无阻碍性。它也确实满足我们在图 6.2中的形式化定义。

但是，我们可以构造出一个均匀隔离的事件路径，它既不是无锁的也不是无等待的。考虑客户端程序 f() ‖ g() 以及图 A.1 中的执行。从 $\mathtt{x} = 0$，$\mathtt{a} = -1$ 且 $\mathtt{b} = 1$ 的状态出发，我们让两个线程轮流执行越来越多次循环迭代。那么，对于每个线程，对于任何 k，我们总是可以找到该线程连续执行的 k 次迭代。因此，该路径是均匀隔离的。但它不满足无锁性或无等待性，因为没有一个方法调用返回。所以，该对象不满足 Herlihy 和 Shavit 的无阻碍性或无冲突性定义。

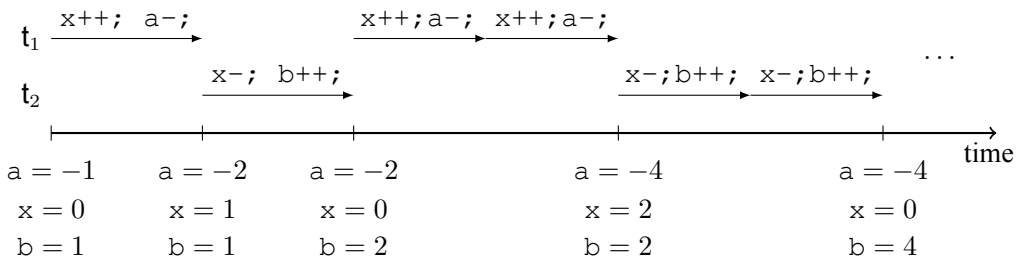


图 A.1 执行示例

附录 B　各种等价结果的证明

本附录简要证明线性一致性、进展性性质以及上下文精化之间的各种等价关系，包括定理 5.6（线性一致性与基本上下文精化之间等价）和定理 6.3（涉及进展性性质的新的等价结果）。更详细的证明过程可见技术报告 [72]。

下面我们用 $\mathcal{T}[\![W,\mathcal{S}]\!]$ 表示 W 在初始状态 $\mathcal{S}$ 下执行生成的有限事件路径的前缀闭集。它扩展了图 5.6 中的 $\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]$ 的定义，允许初始状态中的调用栈非空。类似地，我们使用 $\mathcal{H}[\![W,\mathcal{S}]\!]$ 和 $\mathcal{O}[\![W,\mathcal{S}]\!]$ 分别表示历史和有限可观测事件路径的集合。对于完整路径，也有类似的记号。

B.1　最泛化客户端程序

我们的证明过程基于*最泛化客户端程序*（most general client，简写为 MGC）。简单来说，MGC 是一个特殊的客户端程序，它自身就能够产生任意客户端程序所有可能的行为。我们可以定义线性一致性、进展性性质和上下文精化的 MGC 版本，证明它们与原始定义之间的关系，这样就将原始定义上的等价关系归约为 MGC 版本上的性质。注意线性一致性、进展性性质和上下文精化的原始定义都用全称量词约束客户端程序，直接证明它们的性质难度较大。而 MGC 是一个特定的客户端程序，证明其上的性质通常要容易些。

下面的证明中，我们定义了三种 MGC，它们产生不同的“泛化”行为。不妨设 $dom(\Pi)=\{f_1,\ldots,f_m\}$。我们引入两种求随机数的原语指令。$x:=\mathbf{rand}(m)$ 将随机数 $i\in[1..m]$ 赋值给 x，$x:=\mathbf{rand}()$ 则计算一个任意的正整数。对于任意 n，我们如下定义 MGC_n。

$$
\begin{aligned}
\mathsf{MGT_t} &\stackrel{\text{def}}{=} \mathbf{while}\ (\mathbf{true})\{\\
&\qquad x_{\mathsf{t}} := \mathbf{rand}();\ y_{\mathsf{t}} := \mathbf{rand}(m);\\
&\qquad z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});\\
&\quad \}\\
\mathsf{MGC}_n &\stackrel{\text{def}}{=} \|_{\mathsf{t}\in[1..n]}\,\mathsf{MGT_t}
\end{aligned}
$$

这里 x_{t}，y_{t} 以及 z_{t} 都是线程 t 的局部变量。MGC_n 的每个线程都持续地随机选择一个参数调用一个方法。类似地，我们定义 MGCp_n，它会打印出方法调用的参数和返回值。

$$
\begin{aligned}
\mathsf{MGTp_t} &\stackrel{\text{def}}{=} \textbf{while}\ (\textbf{true})\{ \\
&\qquad x_{\mathsf{t}} := \textbf{rand}();\ y_{\mathsf{t}} := \textbf{rand}(m);\ \textbf{print}(y_{\mathsf{t}}, x_{\mathsf{t}}); \\
&\qquad z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});\ \textbf{print}(z_{\mathsf{t}}); \\
&\quad \} \\
\mathsf{MGCp}_n &\stackrel{\text{def}}{=} \|_{\mathsf{t}\in[1..n]}\, \mathsf{MGTp_t}
\end{aligned}
$$

第三种 MGC 总是在方法调用结束时打印 1。它对观测对象的进展性很有帮助。

$$
\begin{aligned}
\mathsf{MGTp1_t} &\stackrel{\text{def}}{=} \textbf{while}\ (\textbf{true})\{ \\
&\qquad x_{\mathsf{t}} := \textbf{rand}();\ y_{\mathsf{t}} := \textbf{rand}(m); \\
&\qquad z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});\ \textbf{print}(1); \\
&\quad \} \\
\mathsf{MGCp1}_n &\stackrel{\text{def}}{=} \|_{\mathsf{t}\in[1..n]}\, \mathsf{MGTp1_t}
\end{aligned}
$$

对于上述 MGC，客户端内存 σ_{MGC} 应包含每个线程的局部变量。

$$
\sigma_{\mathsf{MGC}} \stackrel{\text{def}}{=} \{x_{\mathsf{t}} \rightsquigarrow \_, y_{\mathsf{t}} \rightsquigarrow \_, z_{\mathsf{t}} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}
$$

B.2　定理 5.6的证明

我们首先定义线性一致性和精化关系的 MGC 版本。

定义 B.1. $\Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$ 当且仅当

$$
\begin{aligned}
&\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, \sigma_a, T.\ T \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \wedge (\varphi(\sigma_o) = \sigma_a) \\
&\Longrightarrow \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd_n (\sigma_{\mathsf{MGC}}, \sigma_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'
\end{aligned}
$$

其中

$$
\Pi_A \rhd_n (\sigma_{\mathsf{MGC}}, \sigma_a, T) \stackrel{\text{def}}{=} T \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!] \wedge \mathsf{seq}(T)\,.
$$

定义 B.2. $\Pi \sqsubseteq_{\varphi} \Pi_A$ 当且仅当

$$
\begin{aligned}
&\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \\
&\Longrightarrow \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.
\end{aligned}
$$

为了证明定理 5.6，我们证明下列引理。

引理 B.3. $\Pi \preceq_{\varphi} \Pi_A \iff \Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$.

引理 B.4. $\Pi \sqsubseteq_{\varphi} \Pi_A \iff \Pi \sqsubseteq_{\varphi} \Pi_A$.

引理 B.5. $\Pi \sqsubseteq_{\varphi} \Pi_A \iff \Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$.

引理 B.3的证明　我们只需证明：任意客户端程序执行生成的历史都能够由 MGC 执行生成，如下引理所示。

引理 B.6 (MGC 的最泛化性)**.** 对于任意 n，$C_1, \ldots, C_n$，σ_c，σ_{MGC} 和 σ_o，我们有 $\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$。

证明. 我们建立客户端程序与 MGC 之间的模拟关系 $\precsim_{\mathsf{MGC}}$，它满足 (B.1)。

对于任意 W_1，$\mathcal{S}_1$，W_2，$\mathcal{S}_2$ 和 e_1，如果 $(W_1, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (W_2, \mathcal{S}_2)$，那么

(1) 如果 $(W_1, \mathcal{S}_1) \xmapsto{e_1}$ **abort** 且 $\mathsf{is\_obj\_abt}(e_1)$，则
存在 T_2 使得 $(W_2, \mathcal{S}_2) \xmapsto{T_2}{}^{*}$ **abort** 且 $e_1 = \mathsf{get\_hist}(T_2)$；

(2) 如果 $(W_1, \mathcal{S}_1) \xmapsto{e_1} (W_1', \mathcal{S}_1')$，则
存在 T_2，W_2' 和 $\mathcal{S}_2'$ 使得 $(W_2, \mathcal{S}_2) \xmapsto{T_2}{}^{*} (W_2', \mathcal{S}_2')$，
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$ 且 $(W_1', \mathcal{S}_1') \precsim_{\mathsf{MGC}} (W_2', \mathcal{S}_2')$。

(B.1)

该模拟关系如下构造。对于任何线程 t，模拟关系的左边若是客户端代码，则右边是 $\mathsf{MGT_t}$；若左边是对象方法代码，则右边也是同样的方法代码。

(1) 左边客户端代码内的每一步执行对应右边 $\mathsf{MGT_t}$ 的零步。

(2) 左边的每次方法调用对应 $\mathsf{MGT_t}$ 用同样的参数调用同样的方法。

(3) 左边方法体内的每一步对应右边同样的一步。

(4) 左边的每次方法返回对应右边同样的方法返回。此时右边会再次执行到 $\mathsf{MGT_t}$。

根据 (B.1)，我们对产生 $\mathcal{H}[\![W_1, \mathcal{S}_1]\!]$ 中事件路径的执行步数进行归纳，可证明：

$$\text{如果 } (W_1, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (W_2, \mathcal{S}_2)\text{，那么 } \mathcal{H}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{H}[\![W_2, \mathcal{S}_2]\!]\text{。}$$

同时我们知道

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \circledcirc)) \precsim_{\mathsf{MGC}} (\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc))$$

成立，因此该引理得证。 □

通过展开 $\Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$ 和 $\Pi \preceq_{\varphi} \Pi_A$ 的定义，应用引理 B.6，我们就可以证明引理 B.3。

引理 B.4的证明

1. $\Pi \sqsubseteq_\varphi \Pi_A \implies \Pi \subseteq_\varphi \Pi_A$：

   我们证明 MGC_n 生成的每个历史都与 MGCp_n 生成的某个可观测路径“等价”，如下引理所示。

   引理 B.7. 对于任意 n，σ_o 和 σ_{MGC}，下述成立。

   (a) 对于任意 T_1，如果 $T_1 \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$，则存则 T_2 使得 $T_2 \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$ 且 $T_1 \approx T_2$。

   (b) 对于任意 T_2，如果 $T_2 \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$，则存在 T_1 使得 $T_1 \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$ 且 $T_1 \approx T_2$。

   其中 $T_1 \approx T_2$ 如下归纳定义：

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{e_1 \approx e_2 \qquad T_1 \approx T_2}{e_1 :: T_1 \approx e_2 :: T_2}$$

$$\frac{}{(\mathsf{t}, f_i, n) \approx (\mathsf{t}, \textbf{out}, (i, n))} \qquad \frac{}{(\mathsf{t}, \textbf{ret}, n) \approx (\mathsf{t}, \textbf{out}, n)}$$

$$\frac{}{(\mathsf{t}, \textbf{obj}, \textbf{abort}) \approx (\mathsf{t}, \textbf{obj}, \textbf{abort})}$$

   证明. 建立 MGC_n 和 MGCp_n 之间的模拟关系。 □

2. $\Pi \subseteq_\varphi \Pi_A \implies \Pi \sqsubseteq_\varphi \Pi_A$：

   证明. 对于任意 n，$C_1, \ldots, C_n$，σ_c，σ_{MGC} 和 σ_o，由引理 B.6可知

$$\begin{aligned}&\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\\ &\subseteq \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]\,.\end{aligned}$$

   设 $\sigma_a = \varphi(\sigma_o)$。由 $\Pi \subseteq_\varphi \Pi_A$ 可知，

$$\begin{aligned}&\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]\\ &\subseteq \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.\end{aligned}$$

   因此，对于任意 T 满足

$$T \in \mathcal{T}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,,$$

   一定存在 T_{MGC} 使得 $\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T_{\mathsf{MGC}})$ 且

$$T_{\mathsf{MGC}} \in \mathcal{T}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]\,.$$

直观上，我们可以将 T 中的对象事件替换为 T_{MGC} 中的对象事件，而保留其他事件以及它们之间的顺序不变。这样得到的事件路径 T' 满足 $\mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T)$。我们证明 T' 可以由对应的抽象客户端程序执行生成，即

$$T' \in \mathcal{T}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a, \circledcirc)]\!].$$

因此该引理得证。

另一种做法是建立三个程序之间的“模拟关系”$\precsim$。这三个程序分别是具体的客户端程序，抽象 MGC 及抽象的客户端程序。$\precsim$ 满足 (B.2)。

$$\begin{array}{l}
\text{对于任意 } W_1\text{，}\mathcal{S}_1\text{，}W_2\text{，}\mathcal{S}_2\text{，}W_3\text{，}\mathcal{S}_3 \text{ 和 } e_1\text{，} \\
\text{如果 } (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)\text{，那么} \\
\quad (1)\ \text{如果 } (W_1, \mathcal{S}_1) \xmapsto{e_1} \textbf{abort}\text{，则存在 } T_3 \text{ 使得} \\
\qquad (W_3, \mathcal{S}_3) \xmapsto{T_3}{}^{*}\ \textbf{abort} \text{ 且 } e_1 = \mathsf{get_obsv}(T_3)\text{；} \\
\quad (2)\ \text{如果 } (W_1, \mathcal{S}_1) \xmapsto{e_1} (W_1', \mathcal{S}_1') \text{ 且 } \mathsf{is_clt}(e_1)\text{，则} \\
\qquad \text{存在 } W_3' \text{ 和 } \mathcal{S}_3' \text{ 使得 } (W_3, \mathcal{S}_3) \xmapsto{e_1} (W_3', \mathcal{S}_3') \text{ 且} \\
\qquad (W_1', \mathcal{S}_1') \precsim (W_2, \mathcal{S}_2; W_3', \mathcal{S}_3')\text{；} \\
\quad (3)\ \text{如果 } (W_1, \mathcal{S}_1) \xmapsto{e_1} (W_1', \mathcal{S}_1') \text{ 且 } \mathsf{is_obj}(e_1)\text{，则} \\
\qquad \text{存在 } T_2\text{，}W_2'\text{，}\mathcal{S}_2'\text{，}T_3\text{，}W_3' \text{ 和 } \mathcal{S}_3' \text{ 使得} \\
\qquad (W_2, \mathcal{S}_2) \xmapsto{T_2}{}^{*} (W_2', \mathcal{S}_2')\text{，}(W_3, \mathcal{S}_3) \xmapsto{T_3}{}^{*} (W_3', \mathcal{S}_3')\text{，} \\
\qquad \mathsf{get_hist}(e_1) = \mathsf{get_hist}(T_2)\text{，}\mathsf{get_obj}(T_2) = T_3 \text{ 且} \\
\qquad (W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')\text{。}
\end{array} \tag{B.2}$$

也就是说，抽象程序 W_3 在客户端代码上的执行与具体程序 W_1 的执行一致，在对象代码上的执行与 W_2（抽象 MGC）的执行一致。这里我们用 $\mathsf{get\_obj}(T)$ 表示 T 中仅由对象事件构成的子路径。由 (B.2)，我们可以证明：

$$\text{如果 } (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)\text{，那么 } \mathcal{O}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{O}[\![W_3, \mathcal{S}_3]\!]\text{。}$$

初始的时候，

$$\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \circledcirc)) \\
\precsim (\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc);\ \textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_a, \circledcirc))
\end{array}$$

成立，这样我们仍证明了该引理。 □

引理 B.5的证明

1. $\Pi \subseteq_{\varphi} \Pi_A \implies \Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$：

   $\Pi \subseteq_{\varphi} \Pi_A$ 的含义是，具体对象 Π 的每个历史都可以由抽象对象 Π_A 产生。因此，要证明具体对象 Π 满足线性一致性，我们只需证明抽象对象 Π_A 对于它本身是线性一致的。注意 Π_A 的每个方法都是原子的。

   引理 B.8 (Π_A 满足线性一致性)**.** 对于任意 n，σ_{MGC}，σ_a 和 T，如果 $T \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$，那么存在 T_c 和 T' 使得 $T_c \in \mathsf{completions}(T)$，$T_c \preceq_{\mathsf{lin}} T'$，$T' \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$ 且 $\mathsf{seq}(T')$。

   证明. 根据生成 T 的执行，我们如下构造另一个执行。我们将每次方法调用推迟到原子执行其方法体的时候，并在执行方法体后立即返回。这样得到的执行能够生成满足要求的历史 T'。 □

2. $\Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A \implies \Pi \subseteq_{\varphi} \Pi_A$：

   我们只需证明每个线性一致的历史都可以由 MGC 使用抽象对象 Π_A 生成。

   引理 B.9 (重排)**.** 对于任意 n，σ_{MGC}，σ_a，T 和 T'，如果 $T \preceq_{\mathsf{lin}} T'$，$\mathsf{seq}(T')$ 且 $T' \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$，那么 $T \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$。

   证明. 我们构造生成 T 的执行。对于并发调用的方法，执行它们的原子方法体的顺序遵照 T' 中方法调用（和紧跟着的返回事件）的顺序。具体的证明需要对 T 的长度归纳。 □

B.3　定理 6.3的证明

对于无锁性，根据定理 5.6，我们只需证明：

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{lock\text{-}free}_{\varphi}(\Pi) \iff \Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A\,.$$

首先，我们定义无锁性的 MGC 版本。

定义 B.10. $\mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$ 当且仅当

$$\forall n, \sigma_{\mathsf{MGC}}, \sigma_o, T.\ T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))$$

我们证明下列引理。

引理 B.11. $\mathsf{lock\text{-}free}_{\varphi}(\Pi) \iff \mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$.

证明. 类似引理 B.3的证明，我们将任意客户端程序的完整事件路径与 MGC_n 的某个事件路径联系起来。 □

引理 B.12. $\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A \implies \Pi \sqsubseteq_{\varphi} \Pi_A$.

证明. 展开定义即得。 □

引理 B.13. $\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A \implies \mathsf{lock\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$.

证明. 我们的证明利用 $\mathsf{MGTp1}_n$。

(1) 对于任意 n,σ_{MGC},σ_a 和 T,如果 $T \in \mathcal{O}_{\omega}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_a, \circledcirc)]\!]$，那么 T 一定是由 $(\_, \textbf{out}, 1)$ 构成的无限事件路径。因此，由 $\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A$ 可知，如果 $T' \in \mathcal{O}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$，那么 T' 也是由 $(\_, \textbf{out}, 1)$ 构成的无限事件路径。

(2) 对于任意 n，σ_{MGC}，σ_o 和 T，如果 $T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$，那么存在 $T' \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}}, \sigma_o, \circledcirc)]\!]$ 使得

   (a) $T' \backslash (\_, \textbf{out}, 1) = T$。也就是说，去掉 T' 中的所有输出事件就得到了 T。

   (b) $\forall i, \mathsf{t}.\ T'(i) = (\mathsf{t}, \textbf{ret}, \_) \Leftrightarrow T'(i+1) = (\mathsf{t}, \textbf{out}, 1)$。也就是说，$T'$ 中的每个输出事件之后都紧跟着一个返回事件。

   由 (1) 可知，$\mathsf{get\_obsv}(T')$ 是由 $(\_, \textbf{out}, 1)$ 构成的无限事件路径。因此，T' 包含了无限个返回事件，T 也是如此。

根据定义 B.10，该引理得证。 □

引理 B.14. $\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{lock\text{-}free}_{\varphi}(\Pi) \implies \Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A$.

证明. 关键是证明 (B.3)。

$$
\begin{aligned}
&\text{对于任意 } n\text{，} C_1, \ldots, C_n\text{，} \sigma_c\text{，} \sigma_o \text{ 和 } \sigma_a \text{ 满足 } \varphi(\sigma_o) = \sigma_a\text{，} \\
&\text{如果 } (\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \cdot\text{，那么} \\
&\text{存在 } T_a \text{ 使得 } (\lfloor \textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{T_a}{}^{\omega} \cdot \text{ 且} \\
&\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)\text{。}
\end{aligned}
\tag{B.3}
$$

其证明与 $\Pi \subseteq_{\varphi} \Pi_A \Longrightarrow \Pi \sqsubseteq_{\varphi} \Pi_A$ 的证明类似。我们将具体程序的无限事件路径 T 中的对象事件替换为 MGC 使用 Π_A 生成的路径中的对象事件。这样得到的 T_a 满足 $\mathsf{get\_obsv}(T_a) = \mathsf{get\_obsv}(T)$。由 $\mathsf{lock\text{-}free}_{\varphi}(\Pi)$ 可以证明，T_a 也一定是无限的。 □

其他进展性性质的相关证明与上述针对无锁性的证明类似，在此省略。详细的证明过程可见技术报告 [72]。

索　引

K

L

M

致　谢

感谢我的导师，冯新宇教授。他的经验和智慧，他的真知灼见，他的好奇心和探索欲，都令我受益匪浅。他也总是愿意给予我情感支持，教导我人生真谛。感谢他不厌其烦地提醒我“Be logical”，“Be professional”，“Be proactive”，也很抱歉这几年我常常没能做到。冯老师是我的良师，也是我的益友。我们合作默契，同甘共苦。我们一起畅想 research 的未来，谈论 community 里的趣事，也倾诉心中的烦恼。如果没有他在研究中的帮助和指导，没有他在生活中的支持和开解，就不可能有我今日的成绩。我的成长归功于冯老师，我还希望可以从他身上学到更多。我一定会学着体谅，渐渐变得成熟有担当，不再期待完美的保护者，学会维护我们之间的关系。谢谢您，冯老师。谢谢您花在我身上的每一分钟，谢谢您对我的容忍。您永远是我的老师。

感谢我的“官方导师”，邵中教授。虽然我们真正一起工作的时间并不长，但他总是愿意帮助我，当我做错事时也不计较。我很感激他的耐心与涵养。

感谢中科大软件安全实验室以及 Yale FLINT 小组的学生和老师们。谢谢你们的友谊和随处可见的智慧。特别感谢一直关心爱护我的陈意云老师，像百科全书一样的蒋信予，常常陪我寻找生活乐趣的王僖，在我难过时愿意收留我的郭宇和李兆鹏。

感谢这个和善的 community，并不因我的懵懂无知和不善表达而轻视我。相反，他们对我的工作的认可和鼓励，常常让我振奋。他们的建议和意见也让我加倍努力地完善自己的工作。感谢 Matthew Parkinson，他那睿智的微笑总是让我放松和鼓舞，我一直盼望可以再见到他。感谢对我连说“Don’t worry”的温柔的 Derek Dreyer，在我做 presentation 前安慰我鼓励我的 Hongseok Yang，将我视作 colleague 的 Viktor Vafeiadis，以及其他所有曾帮助我鼓励我的同行学者们，谢谢你们，我很想念你们。此外，很感谢曾经审阅本文各部分工作的审稿人们。

本论文谨献给我的姥姥，她于 2014 年 5 月 23 日逝世，享年 87 岁。

最后，谢谢爸爸妈妈。我爱你们。

梁红瑾

2014 年 5 月 27 日

在读期间发表的学术论文与取得的研究成果

已发表论文：

1. Liang H, Feng X, Fu M. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. ACM Trans. Program. Lang. Syst., 2014, 36(1):3:1–3:55.

2. Liang H, Hoffmann J, Feng X, Shao Z. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In Proceedings of 24th International Conference on Concurrency Theory (CONCUR 2013), Buenos Aires, Argentina, pages 227–241, August 2013.

3. Liang H, Feng X. Modular Verification of Linearizability with Non-Fixed Linearization Points. In Proceedings of 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013), Seattle, USA, pages 459–470, June 2013.

4. Liang H, Feng X, Fu M. A Rely-Guarantee-based Simulation for Verifying Concurrent Program Transformations. In Proceedings of 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012), Philadelphia, USA, pages 455–468, January 2012.

5. 梁红瑾, 张昱, 陈意云, 李兆鹏, 华保健. 处理指针相等关系不确定的指针逻辑. 软件学报,2010,21(2):334-343.