Foundations and Trends® in Programming Languages

Progress of Concurrent Objects

Suggested Citation: Hongjin Liang and Xinyu Feng (2020), "Progress of Concurrent Objects", Foundations and Trends® in Programming Languages: Vol. 5, No. 4, pp 282–414. DOI: 10.1561/2500000041.

Hongjin Liang
State Key Laboratory for Novel Software Technology
Nanjing University
China
hongjin@nju.edu.cn

Xinyu Feng
State Key Laboratory for Novel Software Technology
Nanjing University
China
xyfeng@nju.edu.cn

now
the essence of knowledge
Boston — Delft

Contents

Progress of Concurrent Objects

Hongjin Liang[1] and Xinyu Feng[2]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, China; hongjin@nju.edu.cn*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, China; xyfeng@nju.edu.cn*

ABSTRACT

Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, starvation-freedom, or deadlock-freedom. These progress properties describe conditions under which a method call is guaranteed to complete. However, they fail to describe how clients are affected, making it difficult to utilize them in layered and modular program verification. Also we lack verification techniques for starvation-free or deadlock-free objects. They are challenging to verify because the fairness assumption introduces complicated interdependencies among progress of threads. Even worse, none of the existing results applies to concurrent objects with partial methods, i.e., methods that are supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired.

In this tutorial we examine the progress properties of concurrent objects. We formulate each progress property (together with linearizability as a basic correctness requirement) in terms of contextual refinement. This essentially gives us progress-aware abstraction for concurrent objects. Thus,

Hongjin Liang and Xinyu Feng (2020), "Progress of Concurrent Objects", Foundations and Trends® in Programming Languages: Vol. 5, No. 4, pp 282–414. DOI: 10.1561/2500000041.

when verifying clients of the objects, we can soundly replace the concrete object implementations with their abstractions, achieving modular verification. For concurrent objects with partial methods, we formulate two new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF). We also design four patterns to write abstractions for PSF or PDF objects under strongly or weakly fair scheduling, so that these objects contextually refine their abstractions. Finally, we introduce a rely-guarantee style program logic LiLi for verifying linearizability and progress *together* for concurrent objects. It unifies thread-modular reasoning about all the six progress properties (wait-freedom, lock-freedom, starvation-freedom, deadlock-freedom, PSF and PDF) in one framework. We have successfully applied LiLi to verify starvation-freedom or deadlock-freedom of representative algorithms such as lock-coupling lists, optimistic lists and lazy lists, and PSF or PDF of lock algorithms.

1

Introduction

A concurrent object consists of shared data and a set of methods which provide an interface for client threads to access the shared data. Linearizability (Herlihy and Wing, 1990) has been used as a standard definition of the correctness of concurrent object implementations. It describes safety and functionality, but has no requirement about termination of object methods. Various progress properties, such as wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom, have been proposed to specify termination of object methods. In their textbook Herlihy and Shavit (2008) give a systematic introduction of these properties.

Although program termination has been an obvious notion for sequential programs, it becomes much more complex in a concurrent setting. Termination of a method call in a thread is affected not only by the sequential behavior of the method code, but also by interference from the environment. Different implementations of the concurrent object methods have different tolerance of the interference. That's why we need these different progress properties.

We give two implementations of a simple counter object in Figure 1.1. The variable `x` (line `0`) is the object data implementing the counter. Figure 1.1(a) and (b) are two different implementations of the `inc`

```
0  int x; //object data
1  inc(){
2    local t, done := false;
3    while(!done){
4      t := x;
5      done := cas(&x, t, t+1);
6    }
7  }
```

(a)

```
8  inc(){
9    lock();
10     x := x+1;
11   unlock();
12 }
```

(b)

Figure 1.1: Implementations of the counter object.

method, which increments the counter. Figure 1.1(a) shows an optimistic implementation. It takes a snapshot `t` of the counter (line 4). The *atomic* **c**ompare-**a**nd-**s**wap (**cas**) command (line 5) compares the current value of `x` with `t`. If they are equal, it atomically sets `x` to `t+1` and returns `true`. Otherwise it does nothing and returns `false`, and the method has to run another round of the loop to roll back and retry the process. Figure 1.1(b) is a lock-based implementation, where the update of the shared variable `x` is protected by a lock. Here we omit the implementation of locks, which will be discussed later.

The different implementations of the `inc` method have different progress properties. We can consider the following client program to see their difference. The formal definitions of progress properties will be discussed later in Section 5.

```
inc()  ||  while(true){ inc(); }
```

If we use the optimistic version in Figure 1.1(a), the call of `inc()` in the left thread may never terminates because the **cas** command at line `5` may always fail due to the infinite number of calls of `inc()` in the right thread. However, whenever we suspend the execution of the right thread, the `inc()` in the left eventually terminates. Therefore we call Figure 1.1(a) a *non-blocking* implementation. Also, since at least one of the call of `inc()` in the whole program terminates, this is a *lock-free* implementation.

If we use the lock-based `inc` in Figure 1.1(b), whether the call of `inc()` in the left thread terminates or not depends on the implementation of the lock. If the lock implementation is fair, the `inc()` on the left always terminates, otherwise it may always fail to acquire the lock and may never terminate. In both cases, if we suspend the right thread when it is executing line `10`, the `inc()` on the left cannot terminate because it can never acquire the lock (which has been taken by the suspended right thread). So we say the lock-based implementation of `inc` is *blocking*. The termination of a method call relies on both the lock algorithm and the fairness of scheduling.

The goal of this tutorial is to help the reader understand the various progress properties of concurrent objects. We formulate each progress property (together with linearizability as a basic correctness requirement) in terms of contextual refinement. This essentially gives us progress-aware abstraction for concurrent objects. We also introduce a program logic LiLi to formally verify progress properties.

1.1 General Motivation

1.1.1 Progress-Aware Abstraction

Progress properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" (Herlihy and Shavit, 2008). They are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients.

In a modular verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π' that consists of equivalent methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π'. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π' will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (e.g., Filipović *et al.*, 2009; Liang and Feng, 2013) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π', if every observable behavior of any client program using Π can also be observed when the client uses Π' instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (i.e., non-termination). Recently, Gotsman and Yang (2011) showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This tutorial studies four commonly used progress properties (wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom) and their relationships to contextual refinements. We show that, when progress properties are taken into account, one may have the corresponding progress-aware contextual refinement to reestablish the equivalence. We give different abstract specifications Π' for different progress properties. The equivalence results allow us to build abstractions for linearizable objects so that safety and progress of the client code can be reasoned about at a more abstract level.

1.1.2 Program Logic for Progress Verification

Recent program logics for verifying concurrent objects either prove only linearizability and ignore the issue of termination (e.g., Derrick *et al.*, 2011; Liang and Feng, 2013; Turon *et al.*, 2013a; Vafeiadis, 2008), or aim for non-blocking progress properties (e.g., da Rocha Pinto *et al.*, 2016; Gotsman *et al.*, 2009; Hoffmann *et al.*, 2013; Liang *et al.*, 2014), which cannot be applied to blocking algorithms that progress only under fair scheduling. The fairness assumption introduces complicated interdependencies among progress properties of threads, making it incredibly more challenging to verify the lock-based algorithms than

their non-blocking counterparts. We will explain the challenges in detail in Subsection 7.1.

It is important to note that, although there has been much work on deadlock detection or deadlock-freedom verification (e.g., Boyapati *et al.*, 2002; Leino *et al.*, 2010; Williams *et al.*, 2005), deadlock-freedom is often defined as a safety property, which ensures the lack of circular waiting for locks. It does not prevent live-lock or non-termination inside the critical section. Another limitation of this kind of work is that it often assumes built-in lock primitives, and lacks support of ad-hoc synchronization (e.g., mutual exclusion achieved using spin-locks implemented by the programmers). The deadlock-freedom we discuss in this tutorial is a liveness property and its definition does not rely on built-in lock primitives. We discuss more related work on liveness verification in Section 8.

In this tutorial we introduce LiLi, a new rely-guarantee style logic for concurrent objects. It unifies verification of linearizability, wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom in one framework (the name LiLi stands for Linearizability and Liveness). In particular, it supports verification of both mutex-based pessimistic algorithms (including fine-grained ones such as lock-coupling lists) and optimistic ones such as optimistic lists and lazy lists. The unified approach allows us to prove *in the same logic*, for instance, the lock-coupling list algorithm is starvation-free if we use fair locks, e.g., ticket locks (Mellor-Crummey and Scott, 1991), and is deadlock-free if regular test-and-set based spin locks (Herlihy and Shavit, 2008) are used.

1.1.3 Concurrent Objects with Partial Methods

However, *none* of the aforementioned progress-related results applies to concurrent objects with partial methods! A method is *partial* if it is supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired. Concurrent objects with partial methods simply do not satisfy any of the aforementioned progress properties, and people do not know how to give progress-aware abstract specifications for them either. The existing studies on progress properties and progress-aware

contextual refinements have been limited to concurrent objects with total specifications.

As an awkward consequence, we cannot treat lock implementations as objects when we study progress of concurrent objects. Instead, we have to treat `lock_acquire` and `lock_release` as *internal* functions of other lock-based objects. Also, without a proper progress-aware abstraction for locks, we have to redo the verification of `lock_acquire` and `lock_release` when they are used in different contexts (Liang and Feng, 2016), which makes the verification process complex and painful. Note that locks are not the only objects with partial methods. Concurrent sets, stacks and queues may also have methods that intend to block. For instance, it may be sensible for a thread attempting to pop from an empty stack to block, waiting until another thread pushes an item. The reasoning about these objects suffers from the same problems too when progress is concerned.

In this tutorial, we specify and verify progress of concurrent objects with partial methods. We define partial starvation-freedom (PSF) and partial deadlock-freedom (PDF) as progress properties for objects with partial methods, and design abstraction patterns under strongly and weakly fair scheduling. We prove that given a linearizable object implementation Π with partial methods, the contextual refinement between Π and its abstraction Π' under a certain kind of fair scheduling is equivalent to PSF/PDF of Π. We also extend the program logic LiLi to support partial specifications and to reason about blocking primitives. It verifies the contextual refinement between Π and Π', which ensures linearizability and the progress property of Π.

1.2 Overview

The goal of this tutorial is to help the reader understand the various progress properties of concurrent objects.

We start with an informal overview of the background in Section 2. We informally describe the traditional four progress properties (wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom), and analyze the challenges in supporting objects with partial methods.

In Section 3, we introduce the basic technical settings. We define a simple object language, and the generation of execution traces from the operational semantics. We also define fairness of scheduling over the traces.

In Section 4, we define linearizability and the basic contextual refinement which is equivalent to linearizability.

In Section 5, we formulate the four traditional progress properties and the two new progress properties for objects with partial methods.

In Section 6, we give the progress-aware contextual refinement and the abstraction theorems.

In Section 7, we present the program logic LiLi and show the examples we have verified.

Finally, we discuss related work in Section 8 and conclude in Section 9.

2

Background

A concurrent object usually satisfies linearizability, a standard safety criterion, and certain progress property, describing when and how method calls of the object are guaranteed to terminate.

In this section, we first give an overview of linearizability. We then introduce the four traditional progress properties (wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom), and explain the need for new progress properties. We also discuss the contextual refinement and the Abstraction Theorems. Finally we introduce rely-guarantee reasoning, explain how to encode linearizability verification and discuss the challenges in progress verification.

2.1 Linearizability

A concurrent object is linearizable, if each method call appears to take effect instantaneously at some moment between its invocation and return (Herlihy and Wing, 1990). Intuitively, linearizability requires the implementation of each method to have the same effect as an atomic specification.

Consider the two implementations of the counter object in Figure 2.1(b) and (d). We assume that every primitive command is executed

```
0  L_initialize(){  l := 0; }

1  L_acq(){
2    local b := false;
3    while(!b){
4      b := cas(&l, 0, cid);
5    }
6  }

7  L_rel(){
8    l := 0;
9  }
```

(a) test-and-set (TAS) lock impl.

```
10  inc(){
11    L_acq();
12      x:=x+1;
13    L_rel();
14  }
```

(b) counter with a TAS lock

```
15  tkL_initialize(){
16    owner := 0; next := 0; }

17  tkL_acq(){
18    local i, o;
19    i := getAndInc(&next);
20    o := owner;
21    while(i!=o){
22      o := owner;
23    }
24  }

25  tkL_rel(){
26    owner := owner + 1;
27  }
```

(c) ticket lock implementation

```
28  inc_tkL(){
29    tkL_acq();
30      x:=x+1;
31    tkL_rel();
32  }
```

(d) counter with a ticket lock

```
INC(){ x:=x+1; }
```

(e) atomic spec. `INC`

Figure 2.1: Counters with locks.

atomically. A counter provides a method `inc` for incrementing the shared data `x`. Both implementations use locks to synchronize the increments. Intuitively they have the same effect as the atomic specification `INC()` in Figure 2.1(e), so they are linearizable.

The locks themselves could also be viewed as standalone objects. For instance, the test-and-set lock object in Figure 2.1(a) provides the methods `L_acq` and `L_rel` for a thread to acquire and release the lock `l`. Here `cid` represents the current thread's ID, which is a positive integer. The counter's implementation code in Figure 2.1(b) can be viewed as a client of this lock object. The lock object is linearizable, because `L_acq` and `L_rel` both update `l` atomically (if they indeed return). They produce the same effects as the atomic operations `L_ACQ` and `L_REL` (defined below), respectively:

```
L_ACQ(){ l := cid; }      L_REL(){ l := 0; }      (2.1.1)
```

However, linearizability does not characterize progress properties of the object implementations. For instance, the following counter object is still linearizable, even if its method never terminates.

```
inc'(){ L_acq(); x:=x+1; L_rel(); while(true) skip; }
```

2.2 Progress Properties

Various progress properties have been proposed for concurrent objects, such as wait-freedom and lock-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which a method call is guaranteed to successfully finish in an execution. The implementation of the counter in Figure 1.1(a) satisfy lock-freedom. The two implementations in Figure 2.1(b) and (d) satisfy deadlock-freedom and starvation-freedom respectively. We assume that every command is executed atomically.

We use the definitions given by Herlihy and Shavit (2011). Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps. We can view that the atomic specification in Figure 2.1(e) is an ideal wait-free implementation in which the increment is done atomically. It is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are

more complex and involve arrays and atomic snapshots (Aspnes and Herlihy, 1990).

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps. Typical lock-free implementations, such as the well-known Treiber stack (Treiber, 1986), HSY elimination-backoff stack (Hendler *et al.*, 2004) and Harris-Michael lock-free list (Harris, 2001; Michael, 2002), use the atomic **c**ompare-**a**nd-**s**wap instruction **cas** in a loop to repeatedly attempt an update until it succeeds. Figure 1.1(a) shows such an implementation of the counter object. It is lock-free, because whenever `inc` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program (2.2.1), the **cas** instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

```
inc();   ||   while(true) inc();          (2.2.1)
```

Wait-freedom and lock-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Informally people often state deadlock-freedom and starvation-freedom in terms of locks and critical sections (a *critical section* is the code segment between a lock-acquire and the matching lock-release). For example, in their textbook, Herlihy and Shavit (2008) say deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom guarantees that every thread attempting to acquire the lock will eventually succeed.

However, as noted by Herlihy and Shavit (2011), the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock and a particular code segment as a critical section. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread holding a lock will eventually release it. This

assumption requires *fair* scheduling, i.e., every thread gets eventually executed. Otherwise, a thread holding a lock may never be scheduled. Then it has no chance to release the lock, even if the critical section terminates (if executed). As a result, all the other threads trying to acquire the lock are permanently blocked.

Following Herlihy and Shavit (2011), we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions.

The counter in Figure 2.1(b) is deadlock-free, because the test-and-set lock (see Figure 2.1(a)) guarantees that eventually *some* thread will succeed in getting the lock via the **cas** instruction at line `4`, and hence the method call of `inc` in that thread will eventually finish. It is not starvation-free, because there might be a thread that continuously fails to acquire the lock. For the following client program (2.2.2), the **cas** instruction executed by the left thread could always fail if the right thread infinitely often acquires the lock.

```
inc(); print(1);   ||   while(true) inc();        (2.2.2)
```

The counter in Figure 2.1(d) implemented with a ticket lock is starvation-free. Figure 2.1(c) shows the details of the ticket lock implementation. It uses two shared variables `owner` and `next`, which are equal initially. The threads attempting to acquire the lock form a waiting queue. In `tkL_acq`, a thread first atomically increments `next` and reads its old value to a local variable `i` (line `19`). It waits until the lock's `owner` equals its ticket number `i` (lines `20-23`), then it acquires the lock. In `tkL_rel`, the thread releases the lock by incrementing `owner` (line `26`). Then the next waiting thread (the thread with ticket number `i+1`, if there is one) can acquire the lock. We can see that the ticket lock implementation ensures the first-come-first-served property, and hence every thread calling `inc_tkL` can eventually acquire the lock and finish its method call.

Progress properties of concurrent objects with partial methods. However, the aforementioned progress properties are all proposed for concurrent objects with total methods only, i.e., methods that should always

return when executed sequentially. They do not apply to objects with partial methods, such as the lock objects in Figure 2.1(a) and (c), which intend to permanently block at certain situations.

Specifically, deadlock-freedom and starvation-freedom do not apply. They allow permanent blocking but only if the scheduling is unfair. Consider the following client program (2.2.3) using the TAS lock in Figure 2.1(a). One of the method calls never finishes.

```
L_acq();  ||  L_acq();
```
(2.2.3)

It shows that the test-and-set lock object does not satisfy the traditional deadlock-freedom or starvation-freedom property we just presented. Neither does the ticket lock object in Figure 2.1(c).

The problem is that `L_acq` intends to block when the lock is not available. The non-termination in the above example (2.2.3) is just the intention of a correct lock implementation; otherwise the lock cannot guarantee mutual exclusion.

As a result, we need new progress properties for objects with partial methods. Moreover, the new progress properties should be able to distinguish the TAS lock and the ticket lock. Consider the following client program (2.2.4).

```
L_acq(); L_rel();  ||  while(true){ L_acq(); L_rel(); }
```
(2.2.4)

The call to `L_acq()` of the left thread may not return under fair scheduling with the TAS lock, but it must return in fair executions with the ticket lock. This shows that the TAS lock and the ticket lock have different progress properties, which we will call *partial deadlock-freedom* (*PDF*) and *partial starvation-freedom* (*PSF*) respectively (formally defined in Section 5).

The relationships between the progress properties form a lattice shown in Figure 2.2 (where the arrows represent implications). For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. PSF and PDF are generalizations of starvation-freedom and deadlock-freedom respectively.

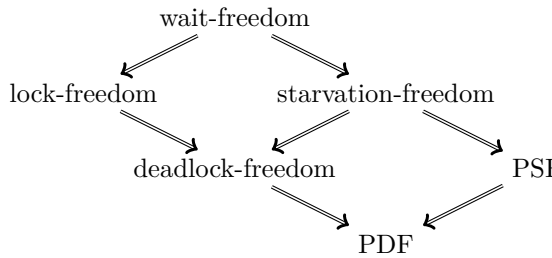

Figure 2.2: Relationships between progress properties.

2.3 Contextual Refinement and Abstraction Theorems

It is difficult to use linearizability and progress properties directly in modular verification of client programs of an object, because their definitions fail to describe how the client behaviors are affected. To verify clients, we would like to abstract away the details of the object implementation. This requires a notion of object correctness, telling us that the client behaviors will not change when we replace the object methods' implementations with the corresponding abstract operations (as specifications).

Contextual refinement gives the desired notion of correctness. Informally, an object implementation Π is a contextual refinement of another (more abstract) implementation Π', written as $\Pi \sqsubseteq \Pi'$, if every observable behavior of any client program using Π can also be observed when the client uses Π' instead. That is,

$$\Pi \sqsubseteq \Pi' \text{ iff } \forall C.\ \mathcal{O}(\textbf{let } \Pi \textbf{ in } C) \subseteq \mathcal{O}(\textbf{let } \Pi' \textbf{ in } C)$$

Here $(\textbf{let } \Pi \textbf{ in } C)$ denotes the client program C using Π, and $\mathcal{O}$ returns the set of observable behaviors of the program. Then, when verifying a client of Π, we can soundly replace Π with its abstraction Π'.

Filipović *et al.* (2009) have proved the abstraction theorem, saying that linearizability is equivalent to a contextual refinement $\Pi \sqsubseteq^{\mathsf{fin}} \Gamma$ between Π and its *atomic specification* Γ. In this contextual refinement, $\mathcal{O}$ returns the prefix-closed set of finite traces of I/O events. A trace set is *prefix-closed* if, for any trace in the set, its prefix must also be in

the set. The superscript in the refinement relation $\sqsubseteq^{\mathsf{fin}}$ means we only observe finite prefixes of execution traces.

This *basic contextual refinement* can distinguish linearizable objects from non-linearizable ones, but it cannot characterize progress properties of objects. For the following example, although no concrete method call of `f` could finish, `f` is a basic contextual refinement of `F`.

```
f(){ while(true) skip; }        F(){ skip; }
```

The reason is that the basic contextual refinement considers a *prefix-closed* set of event traces at the abstract side. For instance, consider the following client (2.3.1).

```
print(1); f(); print(1);                    (2.3.1)
```

The prefix-closed set of externally observable event traces of the client calling `f` is $\{\epsilon, 1\}$, while the prefix-closed set produced by the same client calling `F` is $\{\epsilon, 1, 11\}$, where ϵ represents the empty trace. The former is a subset of the latter. But if we consider only *complete* traces instead of also taking prefixes, we will break the subset relation. The client calling `f` produces a singleton set $\{1\}$ of complete I/O traces, while calling `F` produces $\{11\}$. The subset no longer holds, telling us that `f` and `F` have different termination behaviors.

When taking progress properties into account, the corresponding contextual refinement should be sensitive to termination or divergence (non-termination). In particular, one should observe only *complete* execution traces of I/O events instead of taking the prefix-closed set of traces. Then, as we have seen, for the above example, the sets $\mathcal{O}$ of observable behaviors of the client (2.3.1) calling `f` and `F` will be different. The termination-sensitive contextual refinement does not hold.

But this is not the end of the story. To formulate a contextual refinement $\Pi \sqsubseteq \Pi'$, we need to fix at least three key aspects: the observable behaviors $\mathcal{O}$, the scheduling and the abstraction Π'.

- First, for multi-threaded client programs, do we observe per-thread termination/divergence or whole-program termination/divergence? Consider the lock-free counter `inc` in Figure 1.1(a) and the client (2.2.1). If we observe per-thread termination/divergence, the client

(2.2.1) using `inc` would have different observable behaviors from the client calling the atomic `INC`, because the former may observe the divergence of the left thread (i.e., the left thread calling `inc` may execute infinitely many steps), while the latter cannot (i.e., the left thread calling `INC` always terminates if executed). However, the client has the same whole-program termination behaviors when using `inc` and `INC`.

- Second, is the scheduling fair or not? Some of the progress properties assume fair scheduling, while others do not. Thus it might be natural that the contextual refinement has the same assumption on the scheduling as the corresponding progress property.

  Actually different assumption of fairness may lead to different contextual refinement results. Consider the deadlock-free counter `inc` in Figure 2.1(b) and the atomic `INC` in Figure 2.1(e). The contextual refinement $\texttt{inc} \sqsubseteq \texttt{INC}$ does not hold if we assume fair scheduling (no matter we observe whole-program termination or per-thread termination), because the client (2.2.2) produces different output events using `inc` and `INC` in fair executions. Using `inc`, it is possible that the left thread in (2.2.2) never prints `1` and the whole program generates an empty trace because the lock acquire in the left thread may always fail, but it must print out `1` if `INC` is called instead and the scheduling is fair — as long as the left thread gets a chance to run, the atomic `INC` will finish in one step. However, if we drop the fairness assumption when the abstract atomic specification `INC` is called, we can use an unfair scheduling that never schedules the left thread to simulate the execution of `inc` where the lock acquire always fails (Liang *et al.*, 2013).

- Finally, are the abstractions atomic or non-atomic? To characterize linearizability, $\Pi \sqsubseteq^{\mathsf{fin}} \Gamma$ directly uses atomic specifications Γ as the abstraction of Π. But to characterize progress properties, the abstractions may have to be non-atomic to simulate the non-terminating behaviors of the concrete implementations under interleavings. As we explain above, the client (2.2.2) produces

different output events using `inc` and `INC` in fair executions. One way to re-establish the refinement is to allow unfair scheduling for the right-hand side of the refinement (i.e., for clients calling `INC`). Another way is to use a non-atomic specification while keeping the assumption of fair scheduling. For instance, the client (2.2.2) cannot distinguish `inc` and the following non-atomic `INC_NA`.

```
INC_NA(){
  while (done) {};
  < x := x + 1; done := true >;
  done := false;
}
```

Here `done` is a newly introduced object variable whose initial value is `false`, and $\langle C \rangle$ is an atomic block. When (2.2.2) uses `INC_NA`, the left thread may not print `1` because the loop `while(done){}` may always fail to terminate if the right thread infinitely often sets `done` to `true`. So (2.2.2) using `inc` and `INC_NA` can produce the same output traces in fair executions. In fact `inc` $\sqsubseteq$ `INC_NA` holds under fair scheduling.

Abstractions for objects with partial methods. The atomic specifications defined in (2.1.1) can characterize the atomic behaviors of lock objects, but they fail to specify that `L_ACQ` should be partial in the sense that it should be blocked when the lock is unavailable.

To address the problem, a natural way is to introduce the *atomic partial specification* Γ, where each method specification is in the form of $\mathbf{await}(B)\{C\}$. The execution of $\mathbf{await}(B)\{C\}$ is blocked if B does not hold, and C executes atomically if B holds (also the current thread cannot be interrupted between the test showing B holds and the execution of C). When B holds, we say $\mathbf{await}(B)\{C\}$ is *enabled*. Note that $\mathbf{await}(B)\{C\}$ is the only *blocking* primitive we introduce in this work. All other primitive commands we have seen so far are non-blocking and they are always enabled. We say a thread is *enabled* if the next command to be executed by the thread is enabled.

For the lock objects, we can define the atomic partial specification Γ as follows.

```
L_ACQ'(){  await (l = 0) { l := cid };  }
L_REL(){ l := 0; }
```
(2.3.2)

The **await** block naturally specifies the atomicity of method functionality, just like the traditional atomic specification $\langle C \rangle$ (which can be viewed as syntactic sugar for **await**(true){C}), therefore Γ may serve as a specification for linearizable objects. It also shows the fact that the object method is partial, with explicit specification of the enabling condition B. Below we will use the atomic partial specification as the starting point to characterize the progress of objects.

Although the atomic partial specification Γ describes the atomic functionality and the enabling condition of each method, it is insufficient to serve as a progress-aware abstraction for the following reasons.

First, the progress of the **await** command itself is affected by the fairness of scheduling, such as strong fairness and weak fairness.

- *Strong fairness*: Every thread which is *infinitely often* enabled will execute infinitely often. Then, **await**(B){C} is not executed only if B is continuously false after some point in the execution trace.
- *Weak fairness*: Every thread which is *eventually always* enabled will execute infinitely often. Then, **await**(B){C} may not be executed when B is infinitely often false. This fairness notion is weaker than strong fairness.

As a result, the choice of fair scheduling will affect the behaviors of a program or a specification with **await** commands. To see this, we consider the following client program (2.3.3).

$[\_]_{\text{ACQ}}$; $[\_]_{\text{REL}}$; print(1); || while(true){ $[\_]_{\text{ACQ}}$; $[\_]_{\text{REL}}$; }
(2.3.3)

where $[\_]_{\text{ACQ}}$ and $[\_]_{\text{REL}}$ represent holes to be filled with method calls of lock acquire and release, respectively. Table 2.1 shows the behaviors of the client with different locks.

If the client calls the abstract specifications in (2.3.2), it must execute `print(1)` under strongly fair scheduling, but may not do so

Table 2.1: Client (2.3.3) with different locks. "Yes" means it must print out 1, "No" otherwise

	Strong Fairness	**Weak Fairness**
spec. (2.3.2)	Yes	No
ticket lock (Fig. 2.1(c))	Yes	Yes
test-and-set lock (Fig. 2.1(a))	No	No

under weakly fair scheduling. This is because the call of `L_ACQ'` by the left thread could be infinitely often enabled and infinitely often disabled in an execution. In detail, whenever the lock is owned by the right thread, `L_ACQ'` on the left is disabled, and whenever the right thread releases the lock, `L_ACQ'` on the left is enabled. In a weakly fair execution, it is possible that the left thread does not execute any step since weak fairness guarantees a thread to execute only when it is *always enabled*. However, such an execution does *not* satisfy strong fairness, which requires the thread to execute infinitely often when it is infinitely often enabled (instead of being always enabled). This example also explains in what sense weak fairness is weaker than strong fairness.

Also note that the two fairness notions coincide when the program does not contain blocking operations and is thus always enabled. Both strong and weak fairness degrade to fairness (saying that every thread executes infinitely often) for programs without **await** commands. Therefore, as shown in Table 2.1, regardless of strongly or weakly fair scheduling, the client (2.3.3) using a ticket lock always executes `print(1)`, but it may not do so if using a test-and-set lock instead. In detail, the ticket lock implementation guarantees that the left thread eventually acquires the lock. But with the test-and-set lock in Figure 2.1(a), it is possible that the **cas** in the lock-acquire method `L_acq()` of the left thread always fails and hence `L_acq()` does not return.

As a result, for the same object implementation, we may need *different abstractions under different scheduling*. As shown in Table 2.1, the specification (2.3.2) cannot serve as the specification of the test-and-set locks under both strong fairness and weak fairness.

Second, even under the same scheduling, *different implementations demonstrate different progress, therefore need different abstractions.*

As shown in Table 2.1, the different lock implementations have different behaviors, even under the same scheduling.

For the above two reasons, we need different abstractions for different combinations of fairness and progress. For PSF and PDF under strong and weak fairness respectively, we may need four different abstractions. Can we generate all of them in a systematic approach?

2.4 Verifying Progress Properties

Below we first give an overview of the traditional rely-guarantee logic for safety proofs (Jones, 1983), which serves as the basis of our concurrency reasoning. Then we show how it can be extended to do relational reasoning and verify the contextual refinement equivalent to linearizability. Our program logic LiLi actually further extends the idea to verify different contextual refinements corresponding to different progress properties and linearizability. Finally we explain the challenges in supporting progress verification, and give a quick look at our ideas of LiLi.

2.4.1 Rely-Guarantee Reasoning

In rely-guarantee reasoning (Jones, 1983), each thread is verified in isolation under some assumptions on its environment (i.e., the other threads in the system). The judgment is in the form of $R, G \vdash \{P\}C\{Q\}$, where the pre- and post-conditions P and Q are assertions over program states and they specify the initial and final states of the program C respectively. The rely condition R and the guarantee condition G are assertions over *state pairs* (i.e., state transitions). The rely condition R specifies the permitted state transitions that the environment threads may have. The guarantee condition G specifies the possible transitions made by the thread C itself. Informally, the judgment $R, G \vdash \{P\}C\{Q\}$ says that, starting from an initial state satisfying P, and assuming that the environment's state transitions all satisfy R, the execution of C is safe, every state transition made by C satisfies G, and the final state (if C terminates) satisfies Q.

The key rule in rely-guarantee reasoning is the following (PAR) rule:

$$\frac{\begin{array}{ll} R_1, G_1 \vdash \{P_1\}C_1\{Q_1\} & G_1 \Rightarrow R_2 \\ R_2, G_2 \vdash \{P_2\}C_2\{Q_2\} & G_2 \Rightarrow R_1 \end{array}}{R_1 \wedge R_2, G_1 \vee G_2 \vdash \{P_1 \wedge P_2\}C_1 \,\|\, C_2\{Q_1 \wedge Q_2\}} \text{(PAR)}$$

Here $C_1 \,\|\, C_2$ is the parallel composition of the two threads C_1 and C_2. The rule says that, we can verify $C_1 \,\|\, C_2$ by separately verifying each thread C_i, showing its behaviors under the rely condition R_i indeed satisfy its guarantee G_i. To ensure that parallel threads can collaborate, the guarantee of each thread needs to satisfy the rely of the other (i.e., $G_1 \Rightarrow R_2$ and $G_2 \Rightarrow R_1$). After parallel composition, the threads should be executed under their common environment (i.e., $R_1 \wedge R_2$) and guarantee all the possible transitions made by them (i.e., $G_1 \vee G_2$).

2.4.2 Relational Reasoning and Linearizability Verification

Consider the counter object `inc` implemented with a test-and-set (TAS) lock in Figure 2.1(b). Verifying linearizability of `inc` requires us to prove that it has the same abstract behaviors as `INC` in Figure 2.1(e), which increments the counter `x` atomically.

Following previous work (Liang and Feng, 2013; Liang *et al.*, 2014; Vafeiadis, 2008), one can extend the rely-guarantee logic to verify contextual refinement between concurrent programs. Since linearizability is equivalent to contextual refinement where the abstract specifications are atomic operations, and, as explained in Subsection 2.3, we wish to establish similar results for the different progress properties, a program logic supporting contextual refinement verification can be applied to verify linearizability and progress properties.

Using the counter object `inc` as an example, the judgment to verify it is in the form of

$$R, G \vdash \{P \wedge \mathsf{arem}(\texttt{INC})\}\texttt{inc}\{Q \wedge \mathsf{arem}(\texttt{skip})\}.$$

It looks similar to the traditional rely-guarantee logic, but we want to do relational reasoning here to show `inc` refines `INC`. The pre- and post-conditions are now *relational* assertions specifying the consistency relation between the program states at the concrete and the abstract

sides. We also use an assertion arem(C) to specify as an auxiliary state (also known as a ghost state) the abstract operation C to be refined by the concrete program. So the precondition says `inc` needs to refine `INC`, and the assertion arem(`skip`) in the postcondition says there is no more abstract operations to be refined at the end of `inc` (so the execution of `inc` indeed fulfills the action `INC`).

Similarly, the rely and guarantee conditions R and G are also lifted to the relational setting. They now specify state transitions at both the concrete and the abstract sides. That is, transitions are made over state pairs consisting of the concrete and abstract states, so R and G are relational assertions over pairs of state pairs.

Readers can refer to previous work (Liang and Feng, 2013; Liang *et al.*, 2014; Vafeiadis, 2008) for more details about relational rely-guarantee logic. In LiLi we extend the relational reasoning to verify contextual refinements equivalent to different progress properties. The focus of this tutorial to address the challenges for progress reasoning, which we first give an overview in the next subsection.

2.4.3 Challenges of Progress Verification

Progress properties of objects such as deadlock-freedom and starvation-freedom have various termination requirements of object methods. They must be satisfied with interference from other threads considered, which makes the verification challenging.

Non-Termination Caused by Interference

In a concurrent setting, an object method may fail to terminate due to interference from its environment. Below we point out there are two different kinds of interference that may cause thread non-termination, namely *blocking* and *delay*. Let's first see a classic deadlocking example.

```
DL-12 :                    DL-21 :
  lock L1; lock L2;          lock L2; lock L1;
  unlock L2; unlock L1;      unlock L1; unlock L2;
```

The methods `DL-12` and `DL-21` may fail to terminate because of the circular dependency of locks. This non-termination is caused by permanent

blocking. That is, when `DL-12` tries to acquire `L2`, it could be blocked if the lock has been acquired by `DL-21`.

The second example is the following client using the lock-free counter in Figure 1.1(a).

```
inc(); || while (true) inc();
```

The call of the `inc` method by the left thread may never terminate. This is because, just before the left thread updates `x` at line `5`, it could be preempted by the right thread, who updates `x` ahead of the left. Then the left thread would fail at the **cas** command and have to loop at least one more round before termination. This may happen infinitely many times, causing non-termination of the `inc` method on the left. In this case we say the progress of the left method is *delayed* by its environment's successful **cas**.

The key difference between blocking and delay is that blocking is caused by the *absence* of certain environment actions, e.g., releasing a lock, while delay is caused by the *occurrence* of certain environment actions, e.g., a successful **cas**. In other words, a blocked thread can progress only if its environment progresses first, while a delayed thread can progress if we suspend the execution of its environment.

Lock-free algorithms disallow blocking (thus they do not rely on fair scheduling), although delay is common, especially in optimistic algorithms. Starvation-free algorithms allow (limited) blocking, but not delay. As the above example of `inc` shows, delay from non-terminating clients may cause starvation. Deadlock-free algorithms allow both (but with restrictions). For instance, consider the TAS lock based counter in Figure 2.1(b) and the following client.

```
inc(); || inc();
```

Suppose the lock is available, and the left thread tries to acquire the lock and is just before the **cas** command in the code of `L_acq` in `inc`. Then it might be *delayed* by the right thread who preempts and gets the lock first. After the right thread acquires the lock, the left thread becomes *blocked* and waits for the lock release by the right thread.

In some algorithms (such as the optimistic list), blocking and delay can be intertwined by the combined use of blocking-based synchronization

```
 1  local b := false, p, c;
 2  while (!b) {
 3    (p, c) := find(e);
 4    lock p; lock c;
 5    b := validate(p, c);
 6    if (!b) {
 7      unlock c; unlock p; }
 8  }
 9  update(p, c, e);
10  unlock c; unlock p;
```

Figure 2.3: The optimistic list.

and optimistic concurrency, which makes the reasoning significantly more challenging than reasoning about lock-free algorithms.

Figure 2.3 shows part of the optimistic list implementation. Each node of the list is associated with a TAS lock. A thread first traverses the list without acquiring any locks (line 3). The traversal `find` returns two adjacent node pointers `p` and `c`, the position where the update (adding or removing elements) to the list will take place. The thread then locks the two nodes (line 4), and calls `validate` to check if the two nodes are still valid list nodes (line 5). If validation succeeds, then the thread performs the update (line 9). Otherwise it releases the two node locks (line 7) and restarts the traversal. We can see that blocking and delay are intertwined in this algorithm—a thread may be blocked when acquiring a lock, and be delayed if the validation fails and the thread has to release the acquired locks and restart the traversal.

How do we come up with general principles to allow blocking and/or delay, but on the other hand to guarantee lock-freedom, starvation-freedom or deadlock-freedom?

Avoid Circular Reasoning

Rely-guarantee style logics provide the power of thread-modular verification by *circular reasoning*. When proving the behaviors of a thread $\mathtt{t}$ guarantee G, we assume that the behaviors of the environment thread $\mathtt{t}'$ satisfy R. Conversely, the proof of thread $\mathtt{t}'$ relies on the assumptions on the behaviors of thread $\mathtt{t}$.

```
 1  lock L1;                  11  lock L2;
 2  local r := L2;            12  local s := L1;
 3  while (r != 0) {          13  while (s != 0) {
 4    unlock L1;              14    unlock L2;
 5    lock L1;                15    lock L2;
 6    r := L2;                16    s := L1;
 7  }                         17  }
 8  lock L2;                  18  lock L1;
 9  unlock L2;                19  unlock L1;
10  unlock L1;                20  unlock L2;
```

Figure 2.4: Rollback with two locks.

However, circular reasoning is usually unsound in liveness verification (Abadi and Lamport, 1995). For instance, we could prove termination of each thread in the deadlocking example above, under the assumption that each environment thread eventually releases the lock it owns. How do we avoid the circular reasoning without sacrificing rely-guarantee style thread-modular reasoning?

The deadlocking example shows that we should avoid circular reasoning to rule out circular dependency caused by blocking. Delay may also cause circular dependency too. The left side of Figure 2.4 shows a thread `t` using two locks. It first acquires `L1` (line `1`) and then tests whether `L2` is available (line `2`). If the test fails, the thread rolls back. It releases `L1` (line `4`), and then repeats the process of acquiring `L1` (line `5`) and testing `L2` (line `6`). Suppose another thread $\mathtt{t}'$ does the opposite (see the right side of Figure 2.4): repeatedly acquiring `L2` and testing `L1`. In this example the acquirement of `L2` by $\mathtt{t}'$ may cause `t` to fail its test of the availability of `L2`. The test could have succeeded if $\mathtt{t}'$ did not interfere, so $\mathtt{t}'$ delays `t`. Conversely, the acquirement of `L1` by `t` may delay $\mathtt{t}'$. Then the two threads can cause each other to continually roll back, and neither method progresses.

Usually when delay is allowed, we need to make sure that the action delaying other threads is a "good" one in that it makes the executing thread progress (e.g., a step towards termination). This is the case with the "benign delays" in the examples of the lock-free counter, the TAS

lock-based counter and the optimistic list. But how do we tell if an action is good or not? The acquirements of locks in Figure 2.4 do seem to be good because they make the threads progress towards termination. How do we prevent such lock acquirements from delaying each other, which causes circular delay?

Ad-Hoc Synchronization and Dynamic Locks

One may argue that the circularity can be avoided by simply enforcing certain orders of lock acquirements, which has been a standard way to avoid "deadlock cycles" (note this is a safety property, as we explained in Subsection 1.1). Although lock orders can help liveness reasoning, it has many limitations in practice.

First, the approach cannot apply for ad-hoc synchronization. For instance, there are no locks in the following deadlocking program.

```
x := 1;                  ||  y := 1;
while (y = 1) skip;      ||  while (x = 1) skip;
x := 0;                  ||  y := 0;
```

Moreover, sometimes we need to look into the lock implementation to prove starvation-freedom. For instance, the counter object in Figure 2.1(b) using a TAS lock is deadlock-free but not starvation-free. If we replace the TAS lock with a ticket lock, as in Figure 2.1(d), the counter becomes starvation-free. We may have to look into the lock implementations to verify the counters. Again, there are actually no locks in the programs if we work at a low abstraction level and look into lock implementations.

Second, it can be difficult to enforce the ordering for fine-grained algorithms on dynamic data structures (e.g., lock-coupling list). Since the data structure is changing dynamically, the set of locks associated with the nodes is dynamic too. To allow a thread to determine dynamically the order of locks, we have to ensure its view of ordering is consistent with all the other threads in the system, a challenge for thread-modular verification. Although dynamic locks are supported in some previous work treating deadlock-freedom as a safety property (e.g., Boyapati *et al.*, 2002; Leino and Müller, 2009), it is unclear how to apply the

techniques for general progress reasoning, with possible combination of locks, ad-hoc synchronization and rollbacks.

2.4.4 A Quick Look at Our Ideas

To conclude this section, we informally explain some of the key ideas of our program logic LiLi. We illustrate the ideas by discussing how to verify lock-freedom of the counter in Figure 1.1(a) and starvation-freedom of the ticket lock-based counter in Figure 2.1(d).

Tokens for Delay

For the lock-free counter in Figure 1.1(a), the termination of the loop (at lines `3-5`) in the `inc` method could be delayed by an environment thread's successful **cas**. The loop may even execute infinite number of iterations in clients like (clt-lf). Nevertheless the counter is lock-free because every successful **cas** in the environment corresponds to a terminating method call of `inc`. So the overall system progresses. The reasoning of the lock-free counter consists of two parts.

1. The loop of the `inc` method must terminate if there are no preemption and delaying actions (such as a successful **cas** in the counter example) from the environment.

2. Every method performs only a *finite* number of delaying actions. Thus executing one delaying action must make the method call move towards termination.

In other words, either a method call is not delayed but progresses, or its environment does delaying actions and progresses. In either case the system as a whole must progress, so the object is lock-free.

In our logic LiLi, we formalize the second part of the above reasoning by introducing tokens as ghost states. We assign a finite number of tokens to each method call, and require that a token must be paid to execute a delaying action (we can mark those delaying actions in the rely and guarantee conditions R and G).

The first part of the above reasoning is about loops. LiLi's rule for loops is in the following form to allow delays:

$$\frac{\begin{array}{c}\text{some well-founded metric decreases at each iteration, unless}\\ \text{delayed (i.e., interfered with a delaying action in } R)\end{array}}{R, G \vdash \{P\}\textbf{while } B \textbf{ do } C\{Q\}} \text{ (L-D)}$$

It simply says that the loop should terminate unless delayed. We prove the termination by finding some well-founded metric M that decreases at each round (like the loop variant in the Hoare logic total correctness rule for loops). But if the environment preempts the execution of the loop and performs a delaying action (indicated by the rely condition R), the metric M is allowed to increase. Remember that the environment would pay a token for the delaying action, so we make sure that the overall system progresses.

Definite Actions for Blocking

Due to the use of a lock, the counter in Figure 2.1(d) can be blocked. Nevertheless it is starvation-free because it uses ticket locks and there is *no permanent blocking* under fair scheduling. The reasoning also has two parts.

1. For the thread holding the lock, the lock release must eventually happen because the critical section (i.e., the code at line `30` in Figure 2.1(d)) terminates.

2. For a blocked thread, it must eventually acquire the lock (i.e., its loop at lines `21-23` in Figure 2.1(c) must terminate) because it waits for a *finite* sequence of lock release actions. The threads requesting the lock form a queue. The thread holding the lock is at the head of the queue, and it is dequeued when it releases the lock. Then, for each blocked thread, the sequence of lock release actions that it waits for gets shorter. Eventually the sequence becomes empty and the thread gets the lock.

Our logic LiLi captures the ideas of the above reasoning. We introduce a novel assertion called a "definite action" $\mathcal{D}$, which models a

thread action that, once enabled, must be eventually finished *regardless of environment interference*. In the example of counter, the definite action $\mathcal{D}$ is the lock release action after acquirement. It is *enabled* when the lock is acquired. We should prove that $\mathcal{D}$ is indeed "definite". This can be done by showing that the critical section terminates (without relying on the progress of any other threads), which is just the first part of the above reasoning.

As in the second part of the above reasoning, we verify the loop at lines `21-23` by showing that the current blocked thread waits for a finite sequence of $\mathcal{D}$-s. The first $\mathcal{D}$ in the sequence is enabled initially (i.e., the head thread of the waiting queue acquires the lock). When the first $\mathcal{D}$ is fulfilled, the next $\mathcal{D}$ in the sequence gets enabled (i.e., the next thread in the waiting queue acquires the lock) and the length of the $\mathcal{D}$-sequence decreases. When the sequence becomes empty, we prove that the current thread can progress and terminate its loop, by finding some well-founded metric that decreases at each round (as in the above rule L-D). We formalize the reasoning in LiLi using a WHILE-rule in the following form:

$$\frac{\begin{array}{c}\text{some well-founded metric decreases at each round, unless} \\ \text{blocked but the length of the } \mathcal{D}\text{-sequence decreases}\end{array}}{\mathcal{D}, R, G \vdash \{P\}\mathbf{while}\ B\ \mathbf{do}\ C\{Q\}}\ (\text{L-B})$$

Since the fulfillment of definite actions does not rely on the progress of other threads, and each thread waits for a finite sequence of definite actions only, we break the circular dependency in rely-guarantee style reasoning of progress properties in the presence of blocking behaviors. Also, the definite actions are semantically specified (just like the rely/guarantee conditions), so we can support ad-hoc synchronization and do not rely on built-in synchronization primitives to enforce ordering of events.

We will present the actual LiLi in detail in Section 7. In particular, we will combine the ideas of definite actions and tokens to support both blocking and delay, and address other challenges.

3

Basic Technical Settings

In this section, we describe the concurrent programming language we use throughout the tutorial. We also define the generations of execution traces and fair traces, which will be used in later sections to formulate linearizability, progress properties and contextual refinement.

3.1 The Language

3.1.1 Syntax

Figure 3.1 shows the syntax of the language. A *program* W consists of an *object declaration* Π and n parallel threads $\hat{C}$ as *clients* sharing the object. To simplify the language, we assume there is only one object in each program. Each Π maps method names f_i to annotated method implementations $(\mathcal{P}_i, x_i, C_i)$, where x_i and C_i are the formal parameter and the method body respectively, and the assertion $\mathcal{P}_i$ is an annotated precondition over the object state to ensure the safe execution of the method. It is defined in Figure 3.2 and is used in the operational semantics explained below. A thread $\hat{C}$ is either a command C, or an **end** flag marking termination of the thread. The commands include the standard ones used in separation logic, where

$$
\begin{array}{rlll}
(\mathit{MName}) & f, g \ldots & & (\mathit{PVar}) \quad x, y, z \ldots \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E + E \mid \ldots \\
(\mathit{BExp}) & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid \neg B \mid \ldots \\
(\mathit{Stmt}) & C & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \\
 & & \mid & x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \mid \mathbf{skip} \\
 & & \mid & x := f(E) \mid \mathbf{return}\ E \mid C; C \\
 & & \mid & \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
 & & \mid & \mathbf{await}(B)\{C\} \\
(\mathit{ODecl}) & \Pi, \Gamma & ::= & \{f_1 \rightsquigarrow (\mathcal{P}_1, x_1, C_1), \ldots, f_n \rightsquigarrow (\mathcal{P}_n, x_n, C_n)\} \\
(\mathit{Prog}) & W & ::= & \mathbf{let}\ \Pi\ \mathbf{in}\ \hat{C}_1 \parallel \ldots \parallel \hat{C}_n \\
(\mathit{Thrd}) & \hat{C} & ::= & C \mid \mathbf{end}
\end{array}
$$

Figure 3.1: Syntax of the programming language.

$x := [E]$ and $[E] := E'$ read and write the heap at the location E respectively, and $x := \mathbf{cons}(E, \ldots, E)$ and $\mathbf{dispose}(E)$ allocate and free memory cells respectively. In addition, we have method call ($x := f(E)$) and return ($\mathbf{return}\ E$) commands. The $\mathbf{print}(E)$ command generates *externally observable events*, which are used to define trace refinements in Section 4. The $\mathbf{await}(B)\{C\}$ command is the only *blocking* primitive in the language. It blocks the current thread if B does not hold, otherwise C is executed *atomically* together with the testing of B. We define the syntactic sugar $\langle C \rangle$ for $\mathbf{await}(\mathsf{true})\{C\}$.

Assumptions. We make the following assumptions to simplify the technical setting. There are no regular function calls in either clients or objects. Therefore $x := f(E)$ can only be executed in client code to call object methods, and $\mathbf{return}\ E$ always returns from object methods to clients. Each object method takes only one argument and each method body ends with a **return** command. Object methods never execute the $\mathbf{print}(E)$ command and therefore do *not* generate external events. The command C in $\mathbf{await}(B)\{C\}$ cannot contain **await**, **print**, and method calls and returns. It cannot contain loops either so that it always terminates.

$$
\begin{array}{rrcl@{\quad}rrcl}
(\mathit{ThrdID}) & \mathsf{t} & \in & \mathit{Nat} & (\mathit{Store}) & s, \mathbb{s} & \in & \mathit{PVar} \rightharpoonup \mathit{Val} \\
(\mathit{Heap}) & h, \mathbb{h} & \in & \mathit{Nat} \rightharpoonup \mathit{Val} & (\mathit{Data}) & \sigma, \Sigma & ::= & (s, h) \\
(\mathit{CallStk}) & \kappa, \mathbb{k} & ::= & \circ \mid (s_l, x, C) & & & & \\
(\mathit{ThrdPool}) & \mathcal{K}, \mathbb{K} & ::= & \{\mathsf{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathsf{t}_n \rightsquigarrow \kappa_n\} & & & & \\
(\mathit{PState}) & \mathcal{S}, \mathbb{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) & (\mathit{LState}) & \varsigma, \delta & ::= & (\sigma_c, \sigma_o, \kappa)
\end{array}
$$

$$
\begin{array}{rrcl@{\quad}rrcl}
(\mathit{ExecCtxt}) & \mathbf{E} & ::= & [\,] \mid \mathbf{E}; C & & & & \\
(\mathit{Pre}) & \mathcal{P} & \in & \mathscr{P}(\mathit{Data}) & (\mathit{AbsFun}) & \varphi & \in & \mathit{Data} \rightharpoonup \mathit{Data}
\end{array}
$$

$$
\begin{array}{rrcl}
(\mathit{Event}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
& & \mid & (\mathsf{t}, \mathbf{out}, n)(\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathsf{t}, \mathbf{term}) \\
& & \mid & (\mathbf{spawn}, n) \\
(\mathit{BIdSet}) & \Delta & \in & \mathscr{P}(\mathit{ThrdID}) \qquad (\mathit{PEvent}) \quad \iota \quad ::= \quad (e, \Delta_c, \Delta_o) \\
(\mathit{ETrace}) & \mathcal{E} & ::= & \epsilon \mid e :: \mathcal{E} \quad \text{(co-inductive)} \\
(\mathit{PTrace}) & T & ::= & \epsilon \mid \iota :: T \quad \text{(co-inductive)}
\end{array}
$$

$$
\mathsf{en}(\hat{C}) \stackrel{\text{def}}{=} \begin{cases} B & \text{if } \exists \mathbf{E}, C'.\ \hat{C} = \mathbf{E}[\mathbf{await}(B)\{C'\}] \\ \mathsf{true} & \text{otherwise} \end{cases}
$$

$$
(\sigma_o, \kappa) \models B \ \text{ iff } \ [\![B]\!]_{((\sigma_o.s) \uplus (\kappa.s_l))} = \mathsf{true} \wedge \kappa \neq \circ
$$

$$
\sigma_c \models B \ \text{ iff } \ [\![B]\!]_{\sigma_c.s} = \mathsf{true}
$$

$$
\begin{array}{l}
\mathsf{btids}(\mathbf{let}\ \Pi\ \mathbf{in}\ \hat{C}_1 \parallel \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) \stackrel{\text{def}}{=} \\
\qquad (\{\mathsf{t} \mid \mathcal{K}(\mathsf{t}) = \circ \wedge \neg(\sigma_c \models \mathsf{en}(\hat{C}_\mathsf{t}))\}, \\
\qquad\ \{\mathsf{t} \mid \mathcal{K}(\mathsf{t}) \neq \circ \wedge \neg((\sigma_o, \mathcal{K}(\mathsf{t})) \models \mathsf{en}(\hat{C}_\mathsf{t}))\}\ )
\end{array}
$$

Figure 3.2: States and event traces.

3.1.2 Operational Semantics

Before describing the operational semantics rules, we first define program states $\mathcal{S}$ in Figure 3.2. We use two sets of notations to represent states at the concrete and the abstract levels respectively.

To ensure that the client code does not touch the object data, in $\mathcal{S}$ we separate the data accessed by clients (σ_c) and by object methods (σ_o). Each σ consists of a store s and a heap h, which map variables and memory locations to values respectively. $\mathcal{S}$ also contains a *thread pool* $\mathcal{K}$ mapping thread IDs t to the corresponding method *call stacks* κ. Recall that the only function call allowed in the language is the method

$$\frac{(\hat{C}_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (\hat{C}'_i, (\sigma'_c, \sigma'_o, \kappa'))) \qquad \mathcal{K}' = \mathcal{K}\{i \rightsquigarrow \kappa'\} \quad \mathsf{btids}(\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}'_i \ldots \parallel \hat{C}_n, (\sigma'_c, \sigma'_o, \mathcal{K}')) = (\Delta_c, \Delta_o)}{(\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}_i \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) \xmapsto{(e, \Delta_c, \Delta_o)} (\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}'_i \ldots \parallel \hat{C}_n, (\sigma'_c, \sigma'_o, \mathcal{K}'))}$$

$$\frac{\hat{C}_i = \textbf{skip} \qquad \mathcal{K}(i) = \circ \qquad \hat{C}'_i = \textbf{end} \qquad e = (i, \textbf{term}) \quad \mathsf{btids}(\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}_i \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) = (\Delta_c, \Delta_o)}{(\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}_i \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) \xmapsto{(e, \Delta_c, \Delta_o)} (\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}'_i \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K}))}$$

$$\frac{(\hat{C}_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} \textbf{abort}}{(\textbf{let } \Pi \textbf{ in } \hat{C}_1 \parallel \ldots \hat{C}_i \ldots \parallel \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) \xmapsto{(e, \emptyset, \emptyset)} \textbf{abort}}$$

Figure 3.3: Operational semantics rules—program transitions.

invocation made by a client and there are no nested function calls, therefore each κ is either empty ($\circ$, which means the thread is executing the *client* code), or contains only *one* stack frame (s_l, x, C), where s_l is the local store for the local variables of the method, x is the (client) variable recording the return value, and C is the continuation (the remaining client code to be executed after the return of the method).

The operational semantics rules consist of three parts, including state transitions made by the whole program (Figure 3.3), by individual threads (Figure 3.4), and by clients or object methods (Figure 3.5).

Figure 3.3 shows that the execution of the program W follows the non-deterministic interleaving semantics, which is defined based on thread transitions defined in Figure 3.4. Each transition over program configurations is labelled with a program event ι, a triple in the form of (e, Δ_c, Δ_o). The event e is generated by thread transitions. As defined in Figure 3.2, (t, f, n) records the invocation of the method f with the argument n in the thread t, and $(\mathsf{t}, \textbf{ret}, n)$ is for a method return with the return value n. $(\mathsf{t}, \textbf{obj})$ and $(\mathsf{t}, \textbf{clt})$ record a regular object step and a regular client step respectively, while $(\mathsf{t}, \textbf{obj}, \textbf{abort})$ and $(\mathsf{t}, \textbf{clt}, \textbf{abort})$ are for aborting of the thread in the object and client code respectively.

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

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

$$\frac{\kappa = (s_l, x, C) \qquad [\![E]\!]_{s_l} = n \qquad s_c' = s_c\{x \rightsquigarrow n\}}{(\mathbf{E}[\,\mathbf{return}\ E\,], ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathsf{t},\mathbf{ret},n)}_{\mathsf{t},\Pi} (C, ((s_c', h_c), \sigma_o, \circ))}$$

$$\frac{\kappa = (s_l, x, C) \qquad [\![E]\!]_{s_l} \textit{ undefined}}{(\mathbf{E}[\,\mathbf{return}\ E\,], ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathsf{t},\mathbf{obj},\mathbf{abort})}_{\mathsf{t},\Pi} \mathbf{abort}}$$

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

$$\frac{(C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} (C', (s_o' \uplus s_l', h_o')) \qquad dom(s_l) = dom(s_l')}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C_1))) \xrightarrow{(\mathsf{t},\mathbf{obj})}_{\mathsf{t},\Pi} (C', (\sigma_c, (s_o', h_o'), (s_l', x, C_1)))}$$

$$\frac{(C, \sigma_c) \longrightarrow_{\mathsf{t}} (C', \sigma_c')}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathsf{t},\mathbf{clt})}_{\mathsf{t},\Pi} (C', (\sigma_c', \sigma_o, \circ))}$$

$$\frac{(C, (s_o \uplus s_l, h_o)) \longrightarrow_{\mathsf{t}} \mathbf{abort}}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C_1))) \xrightarrow{(\mathsf{t},\mathbf{obj},\mathbf{abort})}_{\mathsf{t},\Pi} \mathbf{abort}}$$

$$\frac{(C, \sigma_c) \longrightarrow_{\mathsf{t}} \mathbf{abort}}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathsf{t},\mathbf{clt},\mathbf{abort})}_{\mathsf{t},\Pi} \mathbf{abort}}$$

Figure 3.4: Selected operational semantics rules—thread transitions.

The output event $(\mathsf{t}, \mathbf{out}, n)$ is generated by the $\mathbf{print}(E)$ command. $(\mathsf{t}, \mathbf{term})$ records the termination of the thread t. We also introduce a special event $(\mathbf{spawn}, n)$, which is inserted at the beginning of each event trace to record the creation of n threads at the beginning of

$$\frac{[\![B]\!]_s = \mathbf{true} \qquad (C,(s,h)) \longrightarrow_{\mathsf{t}}^{*} (\mathbf{skip},(s',h'))}{(\mathbf{E}[\,\mathbf{await}(B)\{C\}\,],(s,h)) \longrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,],(s',h'))}$$

$$\frac{[\![B]\!]_s = \mathbf{true} \qquad (C,(s,h)) \longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}}{(\mathbf{E}[\,\mathbf{await}(B)\{C\}\,],(s,h)) \longrightarrow_{\mathsf{t}} \mathbf{abort}}$$

Figure 3.5: Selected operational semantics rules—local thread transitions.

the whole program execution. It will be used for defining fairness of scheduling in Subsection 3.2.

The sets Δ_c and Δ_o in the label record the IDs of threads that are blocked in the client code and object methods respectively. They are generated by the function btids defined in Figure 3.2. Recall that a thread t is executing the client code if its call stack is empty, i.e., $\mathcal{K}(\mathsf{t}) = \circ$. We also define $\mathsf{en}(\hat{C})$ as the enabling condition for $\hat{C}$, which ensures that $\hat{C}$ can execute at least one step unless it has terminated. Here the execution context $\mathbf{E}$ defines the position of the command to be executed next.

The second rule in Figure 3.3 shows that **end** is used as a flag marking the termination of a thread. A termination event $(\mathsf{t}, \mathbf{term})$ is generated correspondingly.

The first two rules in Figure 3.4 show that method calls can only be executed in the client code (i.e., when the stack κ is empty), and it is the clients' responsibility to ensure that the precondition $\mathcal{P}$ (defined in Figure 3.2) of the method holds over the object data. If $\mathcal{P}$ does not hold, the method invocation step aborts. Similarly, as shown in the subsequent rules, the **return** command can only be executed in the object method, and the **print** command can only be in the client code. Other commands can be executed either in the client or in the object, and the transitions are made over σ_c and σ_o respectively.

In Figure 3.5 we show the operational semantics for $\mathbf{await}(B)\{C\}$. It tests B and execute C atomically if B holds. Note that there is no transition rule when B is false, which means that the thread is blocked. Transition rules of other commands are standard and omitted here.

More discussions about partial methods. There are actually two reasons that make a method partial. The first is due to non-termination when the method is called under certain conditions. The second is due to abnormal termination, i.e., the method aborts or terminates with incorrect states or return values. Since the goal of this tutorial is to study progress, we focus on the first kind of partial methods.

In our language, we specify the two kinds of partial methods differently. For the first kind, we use the enabling condition B in $\mathbf{await}(B)\{C\}$ to specify when the method should *not* be blocked. For the second kind, we use the annotated precondition $\mathcal{P}$ to specify the condition needed for the method to execute safely and to generate correct results. For instance, although the lock's release method `L_REL` in specification (2.3.2) always terminates, it needs an annotated precondition `l=cid` to prevent client threads not owning the lock from releasing it.

3.2 Execution Traces and Fairness of Scheduling

An *event trace* $\mathcal{E}$ is a (possibly infinite) sequence of events, and a *program trace* T is a (possibly infinite) sequence of labels ι. We use ++ to concatenate two traces, and write $[e_1, \ldots, e_n]$ (or $[\iota_1, \ldots, \iota_n]$) for a trace consisting of a sequence of elements.

In Figure 3.6, we define $\mathcal{T}[\![W, \mathcal{S}]\!]$ as the *prefix closed* set of finite traces T generated during the execution of $(W, \mathcal{S})$. The trace set $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ contains the (possibly infinite) *whole* execution traces T generated by $(W, \mathcal{S})$ but with a special label $((\mathbf{spawn}, |W|), \Delta_c, \Delta_o)$ inserted at the beginning. Here we use $|W|$ to represent the number of threads in W. The event $(\mathbf{spawn}, |W|)$ is used to define fairness, as explained below. Δ_c and Δ_o records the threads blocked in clients and object methods respectively (see the definition of btids in Figure 3.2). At the beginning of an execution, Δ_o must be an empty set since no threads are in method calls. The whole execution trace T may be generated under three cases, i.e., the execution of $(W, \mathcal{S})$ diverges, aborts or gets stuck (terminates or is blocked). We write $(W, \mathcal{S}) \xrightarrow{T}^{\omega} \cdot$ for an infinite execution. In this case, the length of T must be infinite, written as $|T| = \omega$.

$$\mathcal{T}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{T \mid \exists W',\mathcal{S}'.\ (W,\mathcal{S}) \xmapsto{T}{}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \xmapsto{T}{}^{*} \mathbf{abort}\}$$
$$\mathcal{H}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ \mathcal{E} = \mathsf{get\_hist}(T) \wedge T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ \mathcal{E} = \mathsf{get\_obsv}(T) \wedge T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$

$$\mathcal{T}_\omega[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{((\mathbf{spawn},|W|),\Delta_c,\Delta_o)::T \mid \mathsf{btids}(W,\mathcal{S}) = (\Delta_c,\Delta_o) \wedge (((W,\mathcal{S}) \xmapsto{T}{}^{\omega} \cdot) \vee ((W,\mathcal{S}) \xmapsto{T}{}^{*} \mathbf{abort}) \vee \exists W',\mathcal{S}'.\ ((W,\mathcal{S}) \xmapsto{T}{}^{*} (W',\mathcal{S}')) \wedge \neg(\exists \iota.\ (W',\mathcal{S}') \xmapsto{\iota} \_))\}$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}(((\mathbf{spawn},n),\Delta_c,\Delta_o)::T) \stackrel{\text{def}}{=} n$$
$$\mathsf{evt}(\iota) \stackrel{\text{def}}{=} e \quad \text{if } \iota = (e,\Delta_c,\Delta_o) \qquad \mathsf{bset}(\iota) \stackrel{\text{def}}{=} \Delta_c \cup \Delta_o \quad \text{if } \iota = (e,\Delta_c,\Delta_o)$$

$$\mathsf{i\text{-}o\text{-}enabled}(\mathsf{t},T) \quad \text{iff} \quad \forall i.\ \exists j \geq i.\ \mathsf{t} \notin \mathsf{bset}(T(j)) \qquad \text{“infinitely often”}$$
$$\mathsf{e\text{-}a\text{-}enabled}(\mathsf{t},T) \quad \text{iff} \quad \exists i.\ \forall j \geq i.\ \mathsf{t} \notin \mathsf{bset}(T(j)) \qquad \text{“eventually always”}$$

$$\mathsf{fair}(T) \quad \text{iff} \quad |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \mathsf{evt}(\mathsf{last}(T|_{\mathsf{t}})) = (\mathsf{t},\mathbf{term}) \vee |(T|_{\mathsf{t}})| = \omega$$
$$\mathsf{sfair}(T) \quad \text{iff} \quad |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \mathsf{evt}(\mathsf{last}(T|_{\mathsf{t}})) = (\mathsf{t},\mathbf{term}) \vee (\mathsf{i\text{-}o\text{-}enabled}(\mathsf{t},T) \implies |(T|_{\mathsf{t}})| = \omega)$$
$$\mathsf{wfair}(T) \quad \text{iff} \quad |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \mathsf{evt}(\mathsf{last}(T|_{\mathsf{t}})) = (\mathsf{t},\mathbf{term}) \vee (\mathsf{e\text{-}a\text{-}enabled}(\mathsf{t},T) \implies |(T|_{\mathsf{t}})| = \omega)$$

Figure 3.6: Trace generation and fairness.

Histories and externally observable event traces. Linearizability and contextual refinement are defined using traces of history events and externally observable events, respectively. We call (t,f,n), $(\mathsf{t},\mathbf{ret},n)$ and $(\mathsf{t},\mathbf{obj},\mathbf{abort})$ *history* events, and $(\mathsf{t},\mathbf{out},n)$, $(\mathsf{t},\mathbf{obj},\mathbf{abort})$ and $(\mathsf{t},\mathbf{clt},\mathbf{abort})$ *externally observable* events. As defined in Figure 3.6, $\mathcal{H}[\![W,\mathcal{S}]\!]$ contains the set of histories projected from traces in $\mathcal{T}[\![W,\mathcal{S}]\!]$. Here $\mathsf{get\_hist}(T)$ returns a subsequence $\mathcal{E}$ consisting of the history events projected from the corresponding labels in T. Similarly $\mathcal{O}[\![W,\mathcal{S}]\!]$

contains the set of externally observable event traces projected from traces in $\mathcal{T}[\![W, \mathcal{S}]\!]$, where $\mathsf{get\_obsv}(T)$ is a subsequence $\mathcal{E}$ consisting of externally observable events only.

Fair traces. We define three kinds of fairness in Figure 3.6. For a trace in which no thread ever blocks, we can define $\mathsf{fair}(T)$, saying that each thread either terminates, or is executed infinitely many times.

A trace T is strongly fair, represented as $\mathsf{sfair}(T)$, if each thread either terminates, or is executed infinitely many times if it is *infinitely often* enabled ($\mathsf{i\text{-}o\text{-}enabled}$). We know a thread is enabled if it is not in the blocked sets Δ_c and Δ_o. $T(j)$ represents the j-th element in the trace T. Similarly, $\mathsf{wfair}(T)$ says that T is a weakly fair trace. It requires that each thread either terminates, or is executed infinitely many times if it is *always* enabled after certain step on the trace ($\mathsf{e\text{-}a\text{-}enabled}$). It is easy to see that sfair is stronger than wfair, and they both coincide with fair for a trace in which no thread ever blocks, as shown in Lemma 3.1.

Lemma 3.1. For any T, both the following hold:

1. $\mathsf{sfair}(T) \implies \mathsf{wfair}(T)$.
2. if $\forall i.\ T(i) = (\_, \emptyset, \emptyset)$, then $\mathsf{fair}(T) \iff \mathsf{sfair}(T) \iff \mathsf{wfair}(T)$.

4

Linearizability and Contextual Refinement

In this section we formally define linearizability (Herlihy and Wing, 1990) of an object Π with respect to its abstract specification Γ. To support partial methods, Γ is an *atomic partial specification* for Π. It has the same syntax with Π (see Figure 3.1), but each method body in Γ is always an await block $\mathbf{await}(B)\{C\}$ (followed by a $\mathbf{return}\ E$ command). Besides, we assume that the methods in Γ are safe, i.e., they never abort.

We also discuss the Abstraction Theorem for linearizability, i.e., linearizability is equivalent to a contextual refinement which relates the externally observable event traces of Π to those of Γ.

4.1 Linearizability

Linearizability is defined using histories. As defined in Subsection 3.2, histories are special event traces only consisting of method invocation, method return, and object faults.

Definition 4.1 formulates linearizable histories. We say a history $\mathcal{E}$ is linearizable with respect to $\mathcal{E}'$, i.e., $\mathcal{E} \preceq^{\mathsf{lin}} \mathcal{E}'$, if they have the same sub-trace when projected over individual threads (projection represented as $\mathcal{E}|_{\mathsf{t}}$), and $\mathcal{E}$ is a permutation of $\mathcal{E}'$ but preserves the

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

$$\frac{}{\mathsf{seq}(\epsilon)} \qquad \frac{\mathsf{is\_inv}(e)}{\mathsf{seq}(e :: \epsilon)} \qquad \frac{\mathsf{match}(e_1, e_2) \quad \mathsf{seq}(\mathcal{E})}{\mathsf{seq}(e_1 :: e_2 :: \mathcal{E})} \qquad \frac{\forall \mathsf{t}.\ \mathsf{seq}(\mathcal{E}|_{\mathsf{t}})}{\mathsf{well\_formed}(\mathcal{E})}$$

$$\frac{\mathsf{well\_formed}(\mathcal{E})}{\mathcal{E} \in \mathsf{extensions}(\mathcal{E})} \qquad \frac{\mathcal{E}' \in \mathsf{extensions}(\mathcal{E}) \quad \mathsf{is\_ret}(e) \quad \mathsf{well\_formed}(\mathcal{E}'{+}{+}[e])}{\mathcal{E}'{+}{+}[e] \in \mathsf{extensions}(\mathcal{E})}$$

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

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

$$\mathsf{completions}(\mathcal{E}) \stackrel{\text{def}}{=} \{\mathsf{truncate}(\mathcal{E}') \mid \mathcal{E}' \in \mathsf{extensions}(\mathcal{E})\}$$

$$\circledcirc \stackrel{\text{def}}{=} \{\mathsf{t}_1 \rightsquigarrow \circ, \ldots, \mathsf{t}_n \rightsquigarrow \circ\}$$

$$\Gamma \rhd (\Sigma, \mathcal{E}) \text{ iff } \exists n, C_1, \ldots, C_n, \sigma_c. \\ (\mathcal{E} \in \mathcal{H}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!]) \wedge \mathsf{seq}(\mathcal{E})$$

Figure 4.1: Auxiliary definitions for linearizability.

order of non-overlapping method calls in $\mathcal{E}'$. Here we use $\mathsf{is\_inv}(e)$ (or $\mathsf{is\_ret}(e_2)$) to represent that e is in the form of (t, f, n) (or $(\mathsf{t}, \mathbf{ret}, n)$).

Definition 4.1 (Linearizable Histories). $\mathcal{E} \preceq^{\mathsf{lin}} \mathcal{E}'$ iff the following hold.

1. $\forall \mathsf{t}.\ \mathcal{E}|_{\mathsf{t}} = \mathcal{E}'|_{\mathsf{t}}$.
2. There exists a bijection $\pi : \{1, \ldots, |\mathcal{E}|\} \to \{1, \ldots, |\mathcal{E}'|\}$ such that $\forall i.\ \mathcal{E}(i) = \mathcal{E}'(\pi(i))$ and
$$\forall i, j.\ i < j \wedge \mathsf{is\_ret}(\mathcal{E}(i)) \wedge \mathsf{is\_inv}(\mathcal{E}(j)) \implies \pi(i) < \pi(j).$$

Definition 4.2 says Π is linearizable with respect to Γ and the state abstraction function φ (see Figure 3.2) if, for any history $\mathcal{E}$ generated by Π with the initial object data σ, the corresponding complete history $\mathcal{E}_c$ is always linearizable with some *sequential* history $\mathcal{E}'$ generated by Γ with initial object data Σ such that $\varphi(\sigma) = \Sigma$. The set $\mathcal{H}[\![W, \mathcal{S}]\!]$ contains histories generated from finite executions, which is defined in Figure 3.6. Other key notations are defined in Figure 4.1. We use $\circledcirc$ to represent the initial thread pool where each thread has an empty call stack. $\mathsf{completions}(\mathcal{E})$ appends matching return events for some pending

invocations in $\mathcal{E}$, and discards the other pending invocations, so that in the resulting trace every invocation has a matching return. We use $\mathsf{tid}(e)$ for the thread ID in e.

Definition 4.2 (Linearizability of Objects)**.** The object implementation Π is linearizable with respect to Γ, written as $\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma$, iff

$$\begin{aligned}&\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma, \mathcal{E}.\\ &\quad \mathcal{E} \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma)\\ &\implies \exists \mathcal{E}_c, \mathcal{E}'.\ (\mathcal{E}_c \in \mathsf{completions}(\mathcal{E})) \wedge (\Gamma \rhd (\Sigma, \mathcal{E}')) \wedge (\mathcal{E}_c \preceq^{\mathsf{lin}} \mathcal{E}')\end{aligned}$$

4.2 Contextual Refinement and Abstraction

Filipović *et al.* (2009) showed that linearizability is equivalent to a contextual refinement. As defined below, Π contextually refines Γ under the state abstraction function φ if, for any clients $C_1 \ldots C_n$ as the execution context, and for any initial object data related by φ, executing Π generates no more externally observable event traces than executing Γ. Here $\mathcal{O}[\![W, \mathcal{S}]\!]$ contains the externally observable event traces generated from finite executions, which is defined in Figure 3.6.

Definition 4.3 (Basic Contextual Refinement)**.** $\Pi \sqsubseteq^{\mathsf{fin}}_{\varphi} \Gamma$ iff

$$\begin{aligned}&\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ (\varphi(\sigma) = \Sigma) \implies\\ &\qquad \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!]\\ &\qquad\qquad \subseteq \mathcal{O}[\![(\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!].\end{aligned}$$

Theorem 4.1 shows the equivalence between linearizability and the contextual refinement. Its proof follows Filipović *et al.* (2009).

Theorem 4.1 (Abstraction for Linearizability)**.**

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \iff \Pi \sqsubseteq^{\mathsf{fin}}_{\varphi} \Gamma.$$

Theorem 4.1 allows us to use $\Pi \sqsubseteq^{\mathsf{fin}}_{\varphi} \Gamma$ to identify linearizable objects. However, as we have explained in Subsection 2.3, we cannot use it to characterize progress properties of objects, because $\Pi \sqsubseteq^{\mathsf{fin}}_{\varphi} \Gamma$ considers *prefix-closed* sets of event traces.

5

Progress Properties

In this section we discuss progress properties. We first formulate the four traditional progress properties, wait-freedom (WF), lock-freedom (LF), starvation-freedom (SF) and deadlock-freedom (DF), for objects whose methods are all total. Then we define the new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF), for objects with partial methods.

5.1 Progress for Objects with Total Methods Only

We define progress as properties over both event traces T and object implementations Π. In this subsection, we assume that the code of Π does not contain blocking primitives. As a result, the code is always enabled.

Definition 5.1 (Progress of Objects)**.** The object Π satisfies the progress property Prog, written as $\mathsf{Prog}_{\varphi}(\Pi)$, iff

$$\forall n, C_1, \ldots, C_n, \sigma, T.\ T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \| \ldots \| C_n), \sigma]\!] \wedge (\sigma \in dom(\varphi)) \implies \mathsf{Prog}(T).$$

In Definition 5.1, we say an object implementation Π has a progress property Prog ($\mathsf{Prog} \in \{\mathsf{WF}, \mathsf{LF}, \mathsf{SF}, \mathsf{DF}\}$) if all its event traces have the property. Here we use $\mathcal{T}_{\omega}$ to generate the event traces (see Figure 3.6).

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = \mathsf{evt}(T(i)) \wedge \mathsf{is\_inv}(e) \wedge \neg\exists j > i.\ \mathsf{match}(e, \mathsf{evt}(T(j)))\ \}$$

$$\mathsf{abt}(T) \quad \text{iff} \quad \exists i.\ \mathsf{is\_abt}(\mathsf{evt}(T(i)))$$

$$\mathsf{prog\text{-}t}(T) \quad \text{iff} \quad \mathsf{pend\_inv}(T) = \emptyset$$

$$\mathsf{prog\text{-}p}(T) \quad \text{iff} \quad \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j > i.\ \mathsf{is\_ret}(\mathsf{evt}(T(j)))$$

$$\mathsf{sched}(T) \quad \text{iff} \quad |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{WF} \quad \text{iff} \quad \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{LF} \quad \text{iff} \quad \mathsf{sched} \implies \mathsf{prog\text{-}p} \vee \mathsf{abt}$$

$$\mathsf{SF} \quad \text{iff} \quad \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{DF} \quad \text{iff} \quad \mathsf{fair} \implies \mathsf{prog\text{-}p} \vee \mathsf{abt}$$

Figure 5.1: Formalizing WF, LF, SF and DF.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 5.1.

Thread progress and program progress. We use $\mathsf{prog\text{-}t}(T)$ in Figure 5.1 to say that every method call in T eventually finishes. It ensures that each individual thread calling a method eventually returns. $\mathsf{prog\text{-}p}(T)$ says that there is always at least one method call that finishes. It ensures that the whole program is making progress. Note that the return event $T(j)$ in prog-p does not have to be a matching return of the pending invocation e. Here $\mathsf{pend\_inv}(T)$ represents the set of method invocation events that do not have matching returns. $T(1..i)$ represents the prefix of T with length i.

Scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In the case when no thread ever blocks, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace does *not* satisfy sched.

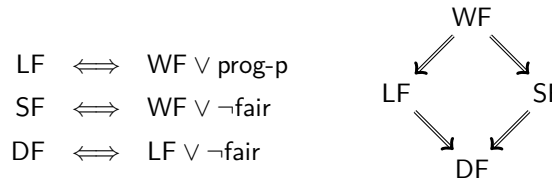

Figure 5.2: Relationships between WF, LF, SF and DF.

$$(\mathrm{t}_1, f_1, n_1) :: (\mathrm{t}_2, f_2, n_2) :: (\mathrm{t}_1, \mathbf{obj}) :: (\mathrm{t}_3, \mathbf{clt}) :: (\mathrm{t}_3, \mathbf{clt}) :: (\mathrm{t}_3, \mathbf{clt}) :: \ldots$$

The fairness of scheduling is defined as $\mathsf{fair}(T)$, $\mathsf{sfair}(T)$ and $\mathsf{wfair}(T)$ in Figure 3.6. We can see $\mathsf{fair} \Rightarrow \mathsf{sched}$, that is, a fair schedule is a good schedule satisfying sched.

Progress properties. At the bottom of Figure 5.1 we define the progress properties over event traces. We omit the parameter T in the formulae to simplify the presentation. An event trace T is wait-free (i.e., $\mathsf{WF}(T)$ holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. $\mathsf{LF}(T)$ is similar except that it guarantees prog-p. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-p under fair scheduling.

Figure 5.2 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice at the right-hand side of the figure, where the arrows represent implications.

5.2 Progress for Objects with Partial Methods

We want to define PSF and PDF as generalizations of starvation-freedom and deadlock-freedom respectively. We say an object Π is *partially starvation-free* if, under fair scheduling (with strong or weak fairness), each method call eventually returns (as required in starvation-freedom), unless it is eventually always *disabled* (i.e., it is not supposed to return in this particular execution context). In the latter case the non-termination is caused by inappropriate invocations of the methods in the client code and the object implementation should *not* be blamed. Similarly, PDF

requires that in each fair execution trace by any client with the object Π, either there always *exists* a method invocation that eventually returns (as in deadlock-freedom), or each pending method invocation must be continuously disabled.

Although the idea is intuitive, it is challenging to formalize it. This is because when we say a method is disabled we are thinking at an abstract level. We actually refer to the enabling condition B in $\mathbf{await}(B)\{C\}$ in the object's atomic partial specification Γ. However, such a condition cannot be syntactically inferred based on the low-level object implementation Π. For instance, the lock implementations in Figure 2.1 use non-blocking commands only, so they are always enabled to execute one more step at this level, although we intend to say at a more abstract level that the `L_acq()` operation is disabled when the lock is unavailable.

To address this problem, we refer to the abstract object specification Γ when defining the progress of a concrete object Π. Recall that method specifications in Γ are in the form of $\mathbf{await}(B)\{C\}$, so we know that the method is disabled when B does not hold.

We formalize the idea as Definition 5.2. Under the scheduling fairness χ (where $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$), we say the object Π is PSF with respect to an abstract specification Γ and a state abstraction function φ, i.e., $\mathsf{PSF}^{\chi}_{\varphi,\Gamma}(\Pi)$, if any χ-fair trace T generated by $((\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc))$ either aborts, or satisfies prog-t, or we could blame the client for the blocking of each pending invocation.

In the last case, we must be able to find a trace T_a generated by the execution of the abstract object Γ (with the abstract object state Σ related to σ by φ) such that it has the *same* method invocation and return history with T, and every pending invocation in this abstract trace T_a is eventually always disabled. See the definition of well-blocked in Figure 5.3 for the formal details.

Definition 5.2 (Partially Starvation-Free Objects)**.** $\mathsf{PSF}^{\chi}_{\varphi,\Gamma}(\Pi)$ iff

$$\begin{aligned}
&\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma, T.\\
&\quad T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma) \wedge \chi(T)\\
&\quad \Longrightarrow\ \mathsf{abt}(T) \vee \mathsf{prog\text{-}t}(T)\\
&\qquad\qquad \vee\ \mathsf{well\text{-}blocked}(T, ((\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \Sigma, \circledcirc))).
\end{aligned}$$

$$\mathsf{e\text{-}a\text{-}disabled}(\mathsf{t}, T) \;\text{ iff }\; \exists i.\, \forall j \geq i, \iota = T(j).\, \mathsf{t} \in \mathsf{bset}(\iota) \qquad \text{“eventually always”}$$

$$\mathsf{well\text{-}blocked}(T, (W_a, \mathcal{S}_a)) \;\text{ iff}$$
$$\exists T_a.\, T_a \in \mathcal{T}_\omega[\![W_a, \mathcal{S}_a]\!] \wedge (\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T_a))$$
$$\wedge\, (\forall e.\, e \in \mathsf{pend\_inv}(T_a) \implies \mathsf{e\text{-}a\text{-}disabled}(\mathsf{tid}(e), T_a))$$

Figure 5.3: Auxiliary definitions for PSF and PDF.

We also define PDF in Definition 5.3. It is similar to PSF, but requires prog-p instead of prog-t.

Definition 5.3 (Partially Deadlock-Free Objects)**.** $\mathsf{PDF}^{\chi}_{\varphi,\Gamma}(\Pi)$ iff

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma, T.$$
$$T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma) \wedge \chi(T)$$
$$\implies \mathsf{abt}(T) \vee \mathsf{prog\text{-}p}(T)$$
$$\vee\, \mathsf{well\text{-}blocked}(T, ((\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc))).$$

The above definitions consider the three factors that may affect the termination of a method call: the scheduling fairness χ, the object implementation Π which determines whether its traces satisfy prog-t or prog-p, and the execution context $C_1 \,\|\, \ldots \,\|\, C_n$ which may make inappropriate method invocations so that well-blocked holds. Consider the lock objects in Figure 2.1(a) and (c) and the following client program (5.2.1). The initial value of the shared variable x is 0.

```
[_]ACQ; print(0); [_]REL;          ||  [_]ACQ; print(2);
x:=1;                              ||  while(x=1){
[_]ACQ; print(1); [_]REL;          ||    [_]REL; [_]ACQ; print(3);
                                   ||  }
```

(5.2.1)

The client can produce a trace satisfying prog-t when it uses the ticket lock. It first executes the left thread until termination and then executes the right thread. Then every method call terminates, printing out 0, 1, 2 and an infinite number of 3. Thus prog-t holds. When the test-and-set lock is used instead, the same client can produce a trace satisfying prog-p but not prog-t. In the execution, the second call to `L_acq` in the left thread never finishes. It prints out 0, 2 and an infinite number

of 3, but not 1. Such an execution is not possible when the client uses the ticket lock, under fair scheduling. This shows how different object implementations affect termination of method calls. Note that neither of the two execution traces satisfies well-blocked, because every method call in the traces either terminates or is enabled infinitely often.

On the other hand, the client (5.2.1) can produce a well-blocked trace no matter it uses the ticket lock or the test-and-set lock. It executes the right thread first until termination and then executes the left thread. Then the first call to lock acquire of the left thread is always blocked, and only 2 is printed during the execution. The non-termination of the method call is caused by the particular execution context, in which the method call is not supposed to return, regardless of the object implementations. This is why the same well-blocked condition is used in both definitions of PSF and PDF for both strongly and weakly fair executions of the object implementation.

PSF (or PDF) and starvation-freedom (or deadlock-freedom) coincide if we require each **await** block in Γ is in the special form of **await**(true){C} —Since the methods in Γ are always enabled, well-blocked$(T, ((\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), \mathcal{S}_a))$ now requires that there is no pending invocation in T. This is stronger than both prog-t(T) and prog-p(T). Therefore we can remove the disjunction branch about well-blocked in Definitions 5.2 and 5.3, resulting in definitions equivalent to starvation-freedom and deadlock-freedom respectively.

6

Progress-Aware Abstraction

In this section we study the abstraction of linearizable objects with the progress properties. Similar to Theorem 4.1, we want theorems showing that linearizability along with a progress property of an object Π is equivalent to a contextual refinement between Π and some abstract object Π', where Π' can be syntactically derived from the atomic specification Γ.

6.1 Overview of Our Results

Contextual refinements for objects with total methods only. Table 6.1 summarizes our results. As we explained in Subsection 2.3, we need to consider three dimensions: the observable behaviors, the scheduling, and the abstractions.

Wait-freedom of linearizable objects is equivalent to a contextual refinement between Π and the atomic specification Γ, where the observable behaviors include the divergence of each individual thread (represented by "(t, Div.)" in Table 6.1) as well as I/O events of (possibly infinite) full executions. The executions can be either fair or unfair (i.e., the scheduling can be "Any" as shown in Table 6.1). As a consequence, if a thread t of a client program diverges when the client uses a linearizable

Table 6.1: Characterizing WF, LF, SF and DF via contextual refinements

	Wait-Free	Lock-Free	Starvation-Free	Deadlock-Free
Observable	(t, Div.)	(t, Div.)	Div.	Div.
Scheduling	Any	Any	Fair	Fair
Abstraction	Atom	Non-Atom	Atom	Non-Atom

and wait-free object Π, then `t` must also diverge when using the atomic specification Γ instead. We say that the abstraction for a wait-free object is atomic ("Atom" in Table 6.1).

Lock-freedom can be characterized by the same contextual refinement except that the abstraction has to be non-atomic (represented by "Non-Atom" in Table 6.1). For instance, such a progress-aware contextual refinement does not hold between the lock-free counter implementation `inc()` in Figure 1.1(a) and the atomic counter `INC` in Figure 2.1(e). The reason is, the left thread of the client program (2.2.1) may diverge if using `inc()` in Figure 1.1(a), but cannot diverge if using the atomic `INC` instead.

Deadlock-freedom or starvation-freedom of linearizable objects is shown equivalent to a contextual refinement which observes divergence (represented by "Div." in Table 6.1) and I/O events of *fair* executions. Then, a client which diverges with Π in a fair execution must also have a diverging execution when using the abstraction Π′ in a fair execution. Deadlock-free and starvation-free objects could be distinguished by different abstractions. The abstraction for starvation-free objects is the atomic specification Γ, while for deadlock-free ones the abstraction has to be non-atomic.

The starvation-free counter `inc_tkL()` in Figure 2.1(d) is a progress-aware contextual refinement of the atomic counter `INC` in Figure 2.1(e), but the deadlock-free `inc()` in Figure 2.1(b) is not. To see the difference, consider the client program (2.2.2). Under fair scheduling, the client calling `inc()` may generate an empty I/O event trace because it may not print out 1. However, the empty trace cannot be generated when replacing `inc()` with `inc_tkL()` or `INC()`, because the resulting program must print out 1.

Table 6.2: Wrappers on atomic specifications for PSF and PDF

	PSF	PDF
Strong Fairness	$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\})$	$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$
Weak Fairness	$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\})$	$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$

Contextual refinements for objects with partial methods. As we have explained in Subsection 2.3, for PSF and PDF under strong and weak fairness respectively, we may need four different abstractions. We define code wrappers over the basic blocking primitive $\mathbf{await}(B)\{C\}$ to generate the abstractions. The code wrappers are syntactic transformations that transform $\mathbf{await}(B)\{C\}$ into possibly non-atomic object specifications which can be refined by the object implementations in the progress-aware contextual refinement. As shown in Table 6.2, the four wrappers correspond to all combinations of fairness and progress. The formal definitions are explained in Subsection 6.5. Here we only give some high-level intuitions using the lock objects as examples.

First, we observe that the lock specification (2.3.2) can already serve as an abstraction for ticket locks under strong fairness, or for test-and-set locks under weak fairness. In general, the wrapper $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}$ can be an identity function, i.e., the atomic partial specifications are already proper abstractions for PSF objects (not only for locks) under strong fairness. But $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}$ is subtle. The atomic partial specifications are insufficient as abstractions for general PDF objects under weak fairness, which we will explain in detail in Subsection 6.5.

Second, as we have seen from Table 2.1, the lock specification (2.3.2) does not work for PSF locks under weak fairness nor for PDF locks under strong fairness. Then the role of the wrapper $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}$ (or $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}$) is to generate the same PSF (or PDF) behaviors even though the fairness of scheduling is weaker (or stronger).

To guarantee PSF, the idea is to create some kind of "fairness" on termination, i.e., every method call can get the chance to terminate. Given weakly fair scheduling, this requires the enabling condition of the abstraction to continuously remain true. As a result, a possible way to define $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\texttt{L\_ACQ'})$ is to keep a queue of threads requesting the

lock, and a thread can acquire the lock only when it is at the head of the queue.

To support PDF under strongly fair scheduling, we have to allow non-termination even if the enabling condition is infinitely often true. For the client (2.3.3), the call of `L_ACQ'` in the specification (2.3.2) under strongly fair scheduling always terminates. Then $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}$ needs to incorporate with some kind of delaying mechanisms, so that the termination of `L_ACQ'` of the left thread could be delayed every time when the right thread succeeds in acquiring the lock.

The abstraction theorem. We prove the abstraction theorem, saying that each progress property P (together with linearizability) is equivalent to a contextual refinement where the abstraction is generated by the corresponding wrapper. On the one hand, the theorem justifies the abstractions generated by our wrappers, showing that they are refined by linearizable object implementations satisfying P. On the other hand, it also justifies our formulation of progress properties (in particular the two new ones, PSF and PDF) by showing that they imply progress-aware contextual refinements.

The abstraction theorem also allows us to verify safety and progress properties of whole programs (consisting of clients and objects) in a modular way. That is, after proving linearizability and progress P of an object Π with respect to its atomic partial specification Γ, we can replace Π with the abstraction generated by applying the corresponding wrapper over Γ, and then reason about properties of the whole program at the high abstraction level.

6.2 Formalizing Progress-Aware Contextual Refinements

We first extend the basic contextual refinement in Definition 4.3 to observe progress as well as linearizability.

Definition 6.1 shows three kinds of progress-aware contextual refinements. The first two, $\Pi \sqsubseteq^{\mathsf{t}\omega}_{\varphi} \Pi'$ and $\Pi \sqsubseteq^{\omega}_{\varphi} \Pi'$, define contextual refinement under arbitrary scheduling. Π contextually refines Π' if, in any execution context, Π generates no more observable behaviors than Π'. The sets of observable behaviors $\mathcal{O}_{\mathsf{t}\omega}[\![W, \mathcal{S}]\!]$ and $\mathcal{O}_{\omega}[\![W, \mathcal{S}]\!]$

are defined in Figure 6.1. $\mathcal{O}_{\omega}[\![W, \mathcal{S}]\!]$ collects the externally observable event traces $\mathcal{E}$ extracted from the full execution traces T in $\mathcal{T}_{\omega}[\![W, \mathcal{S}]\!]$. $\mathcal{O}_{\mathsf{t}\omega}[\![W, \mathcal{S}]\!]$ observes the divergence of individual threads as well as the externally observable event traces. We use $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. These two contextual refinements will be used to characterize wait-free and lock-free objects.

The last kind of contextual refinement in Definition 6.1, $\Pi \sqsubseteq^{\chi}_{\varphi} \Pi'$, defines the contextual refinement for objects under different fairness χ of scheduling ($\chi \in \{\mathsf{fair}, \mathsf{sfair}, \mathsf{wfair}\}$). The set of event traces $\mathcal{O}_{\chi}[\![W, \mathcal{S}]\!]$ is defined at the bottom of Figure 6.1, where each event trace $\mathcal{E}$ is extracted from the χ-fair trace T in $\mathcal{T}_{\omega}[\![W, \mathcal{S}]\!]$. This kind of contextual refinement will be used to characterize SF and DF ($\chi = \mathsf{fair}$), and PSF and PDF ($\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$).

Definition 6.1 (Progress-Aware Contextual Refinement)**.**

$$
\begin{aligned}
&\Pi \sqsubseteq^{\mathsf{t}\omega}_{\varphi} \Pi' \ \text{ iff} \\
&\quad \forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ \ \varphi(\sigma) = \Sigma \implies \\
&\qquad \mathcal{O}_{\mathsf{t}\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \\
&\qquad\quad \subseteq \mathcal{O}_{\mathsf{t}\omega}[\![(\mathbf{let}\ \Pi'\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!] \\
&\Pi \sqsubseteq^{\omega}_{\varphi} \Pi' \ \text{ iff} \\
&\quad \forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ \ \varphi(\sigma) = \Sigma \implies \\
&\qquad \mathcal{O}_{\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \\
&\qquad\quad \subseteq \mathcal{O}_{\omega}[\![(\mathbf{let}\ \Pi'\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!] \\
&\Pi \sqsubseteq^{\chi}_{\varphi} \Pi' \ \text{ iff} \\
&\quad \forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ \ \varphi(\sigma) = \Sigma \implies \\
&\qquad \mathcal{O}_{\chi}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \\
&\qquad\quad \subseteq \mathcal{O}_{\chi}[\![(\mathbf{let}\ \Pi'\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!]
\end{aligned}
$$

Here $\chi \in \{\mathsf{sfair}, \mathsf{wfair}, \mathsf{fair}\}$.

The refinements in Definition 6.1 are *progress-aware* because we use the whole execution trace T here, from which we can tell whether the corresponding execution terminates or not.

Below we discuss the Abstraction Theorem for each progress property. As explained in Subsection 6.1, we define wrappers for atomic (possibly partial) specifications Γ, which transform the method specification

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\,|(T|_{\mathsf{t}})| = \omega\,)\}$$

$$\mathcal{O}_{\mathsf{t}\omega}[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathcal{E}, ts) \mid \exists T.\ T \in \mathcal{T}_\omega[\![W, \mathcal{S}]\!] \wedge \mathsf{get\_obsv}(T) = \mathcal{E} \wedge \mathsf{div\_tids}(T) = ts\}$$

$$\mathcal{O}_\omega[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ T \in \mathcal{T}_\omega[\![W, \mathcal{S}]\!] \wedge \mathsf{get\_obsv}(T) = \mathcal{E}\}$$

$$\mathcal{O}_\chi[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ T \in \mathcal{T}_\omega[\![W, \mathcal{S}]\!] \wedge \chi(T) \wedge \mathsf{get\_obsv}(T) = \mathcal{E}\} \quad \text{where } \chi \in \{\mathsf{fair}, \mathsf{sfair}, \mathsf{wfair}\}$$

Figure 6.1: Observable behaviors.

await$(B)\{C\}$ in Γ into a (possibly non-atomic) abstract specification for each traditional progress property (WF, LF, SF and DF), and for each combination of the two new progress property (PSF or PDF) and fairness (sfair or wfair).

Before introducing the definition of the wrappers in Figures 6.2 and 6.3, we first show our abstraction theorem (Theorem 6.1). It establishes the equivalence between the progress-aware contextual refinement and linearizability with each progress property.

Theorem 6.1 (Abstraction Theorem)**.**

- Let $\mathsf{Prog} \in \{\mathsf{WF}, \mathsf{LF}\}$. Then,

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \wedge \mathsf{Prog}_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{\mathsf{t}\omega}_{\widehat{\varphi}} \mathsf{wr}_{\mathsf{Prog}}(\Gamma).$$

- Let $\mathsf{Prog} \in \{\mathsf{SF}, \mathsf{DF}\}$. Then,

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \wedge \mathsf{Prog}_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{\mathsf{fair}}_{\widehat{\varphi}} \mathsf{wr}_{\mathsf{Prog}}(\Gamma).$$

- Let $\mathsf{Prog} \in \{\mathsf{PSF}, \mathsf{PDF}\}$ and $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$, then

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \wedge \mathsf{Prog}^{\chi}_{\varphi,\Gamma}(\Pi) \iff \Pi \sqsubseteq^{\chi}_{\widehat{\varphi}} \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma).$$

Here $\widehat{\varphi} = \mathsf{wr}_{\mathsf{Prog}}(\varphi)$ if $\mathsf{Prog} \in \{\mathsf{WF}, \mathsf{LF}, \mathsf{SF}, \mathsf{DF}\}$, and $\widehat{\varphi} = \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\varphi)$ if $\mathsf{Prog} \in \{\mathsf{PSF}, \mathsf{PDF}\}$. The wrappers for Γ and φ are defined in Figures 6.2

$$
\begin{array}{rcl}
\mathsf{wr}_{\mathsf{Prog}}(\Gamma)(f) & \stackrel{\text{def}}{=} & (\mathcal{P}, x,\ \mathsf{wr}_{\mathsf{Prog}}(\langle C\rangle); \textbf{return}\ E) \\
& & \qquad \text{if } \Gamma(f) = (\mathcal{P}, x,\ \langle C\rangle; \textbf{return}\ E) \\
\mathsf{wr}_{\mathsf{WF}}(\langle C\rangle) & \stackrel{\text{def}}{=} & \langle C\rangle \\
\mathsf{wr}_{\mathsf{LF}}(\langle C\rangle) & \stackrel{\text{def}}{=} & \texttt{flip}; \langle C\rangle; \texttt{flip}; \\
\texttt{flip} & \stackrel{\text{def}}{=} & \texttt{local tmp, b := false;} \\
& & \texttt{while (!b) \{} \\
& & \quad \texttt{tmp := flag;} \\
& & \quad \texttt{b := cas(\&flag, tmp, !tmp);} \\
& & \texttt{\}} \\
\mathsf{wr}_{\mathsf{SF}}(\langle C\rangle) & \stackrel{\text{def}}{=} & \langle C\rangle \\
\mathsf{wr}_{\mathsf{DF}}(\langle C\rangle) & \stackrel{\text{def}}{=} & \textbf{while}\ (\mathsf{done})\{\}; \\
& & \langle C; \mathsf{done} := \mathsf{true};\rangle;\quad \mathsf{done} := \mathsf{false}; \\
& & \textbf{while}\ (\mathsf{done})\{\}; \\
\mathsf{wr}_{\mathsf{WF}}(\varphi) & \stackrel{\text{def}}{=} & \varphi \\
\mathsf{wr}_{\mathsf{LF}}(\varphi)(\sigma) & \stackrel{\text{def}}{=} & \begin{cases} \sigma' \uplus \{\texttt{flag} \rightsquigarrow \textbf{false}\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases} \\
\mathsf{wr}_{\mathsf{SF}}(\varphi) & \stackrel{\text{def}}{=} & \varphi \\
\mathsf{wr}_{\mathsf{DF}}(\varphi)(\sigma) & \stackrel{\text{def}}{=} & \begin{cases} \sigma' \uplus \{\texttt{done} \rightsquigarrow \textbf{false}\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases}
\end{array}
$$

Figure 6.2: Definition of wrappers for WF, LF, SF and DF.

and 6.3. We also assume that the variables `flag`, `listid` and `done` introduced in the wrapper code are fresh, i.e., $\texttt{flag}, \texttt{listid}, \texttt{done} \notin \mathsf{FV}(\{\Pi, \Gamma, \varphi\})$.

In the following subsections, we discuss the abstraction for each progress property by introducing the definitions of the wrappers in detail.

6.3 Abstraction for Wait-Free and Lock-Free Objects

The contextual refinement for wait-free and lock-free objects takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract sides, where the divergence of

$$\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma)(f) \stackrel{\text{def}}{=} (\mathcal{P}, x, \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\mathbf{await}(B)\{C\}); \mathbf{return}\ E) \quad \text{if } \Gamma(f) = (\mathcal{P}, x, \mathbf{await}(B)\{C\}; \mathbf{return}\ E)$$

$$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\}) \stackrel{\text{def}}{=} \mathbf{await}(B)\{C\}$$

$$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\}) \stackrel{\text{def}}{=} \begin{array}{l} \texttt{listid} := \texttt{listid++}[(\texttt{cid}, \text{`}B\text{'})]; \\ \mathbf{await}(B \wedge \texttt{cid} = \mathbf{enhd}(\texttt{listid}))\{ \\ \quad C;\ \texttt{listid} := \texttt{listid}\backslash\texttt{cid}; \\ \} \end{array}$$

$$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\}) \stackrel{\text{def}}{=} \begin{array}{l} \mathbf{while}\ (\texttt{done})\{\}; \\ \mathbf{await}(B \wedge \neg\texttt{done})\{C; \texttt{done} := \mathsf{true};\}; \\ \texttt{done} := \mathsf{false}; \\ \mathbf{while}\ (\texttt{done})\{\}; \end{array}$$

$$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\}) \stackrel{\text{def}}{=} \begin{array}{l} \mathbf{await}(B \wedge \neg\texttt{done})\{C; \texttt{done} := \mathsf{true};\}; \\ \texttt{done} := \mathsf{false}; \\ \mathbf{await}(\neg\texttt{done})\{\} \end{array}$$

$$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\varphi) \stackrel{\text{def}}{=} \varphi$$

$$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\varphi)(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma' \uplus \{\texttt{listid} \rightsquigarrow \epsilon\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases}$$

$$\mathsf{wr}^{\chi}_{\mathsf{PDF}}(\varphi)(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma' \uplus \{\texttt{done} \rightsquigarrow \mathbf{false}\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases}$$

Figure 6.3: Definition of wrappers for PSF and PDF.

individual threads as well as observable events are treated as observable behaviors. The abstract specifications are generated using the wrappers $\mathsf{wr}_{\mathsf{WF}}$ and $\mathsf{wr}_{\mathsf{LF}}$ in Figure 6.2.

Wrapper for WF. The wrapper $\mathsf{wr}_{\mathsf{WF}}$ is simply an identity function. It maps the atomic (total) specification $\langle C \rangle$ to itself. Thus the abstraction for wait-free objects is the atomic Γ itself. Since a wait-free object Π guarantees that every method call finishes, it behaves just like an atomic object. In particular, the sets div_tids of diverging threads are the same when the client uses Π and Γ. The reason is, we have to blame the client code itself for the divergence of a thread using Π. Thus, even if the thread uses the abstract object Γ, it must still diverge.

As an example, consider the client program (6.3.1).

$$\texttt{inc();} \quad \| \quad \texttt{while(true) inc();} \tag{6.3.1}$$

Intuitively, for any execution in which the client uses the abstract atomic operation `INC`, only the right thread t_2 diverges. Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathsf{t}_2\})\}$. When the client uses a wait-free object, its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathsf{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the lock-free one in Figure 1.1(a)), the left thread t_1 does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing $\mathsf{div\_tids}$ that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

Wrapper for LF. As the example (6.3.1) shows, the set $\mathsf{div\_tids}$ of diverging threads when the client uses a lock-free object Π may be larger than the set $\mathsf{div\_tids}$ when the client uses Γ. To address the problem, we need the wrapper $\mathsf{wr}_{\mathsf{LF}}$ to delay the termination of the abstract atomic operations.

Our first attempt is to introduce a new object variable `flag` and let the wrapper $\mathsf{wr}_{\mathsf{LF}}$ transform $\langle C \rangle$ into:

$$\texttt{flip;}\ \langle C \rangle; \tag{6.3.2}$$

Here the code snippet `flip` is defined in Figure 6.2. It is implemented in a similar way as the lock-free counter (Figure 1.1(a)), using the **cas** command. Therefore the execution of $\mathsf{wr}_{\mathsf{LF}}(\langle C \rangle)$ may not terminate, because `flag` can be flipped infinitely often when other threads continuously finish the wrapped abstract operations. As a result, the set $\mathsf{div\_tids}$ of diverging threads in such an execution can be the same as $\mathsf{div\_tids}$ in the corresponding execution when the client uses the lock-free object. For instance, when the client (6.3.1) uses $\mathsf{wr}_{\mathsf{LF}}(\texttt{INC})$, the $\mathcal{O}_{t\omega}$ set is $\{(\epsilon, \{\mathsf{t}_2\}), (\epsilon, \{\mathsf{t}_1, \mathsf{t}_2\})\}$, which is the same as the client using the lock-free counter.

Note that if the code (6.3.2) fails to terminate, C must not be executed and no effects (over the object data) are generated. However,

it is possible for lock-free methods to finish C and make the effects visible to other threads but fail to terminate. For instance, we define `inc'` in (6.3.3) as a new implementation of the counter.

```
inc'(){ inc(); flip; }                    (6.3.3)
```

It calls the lock-free `inc` method and then execute `flip` defined in Figure 6.2. It is easy to see that `inc'` is still lock-free. The code of `flip` does not change the object variable `x` of the original counter. Its only purpose is to allow the method to diverge after `inc` takes effects.

To simulate this kind of divergence, we insert the code snippet `flip` after the atomic operation $\langle C \rangle$ as well as before it. The resulting wrapper $\mathsf{wr}_{\mathsf{LF}}(\langle C \rangle)$ is shown in Figure 6.2.

Additional Abstraction Theorem for LF. The atomic specification Γ can serve as the abstraction for lock-free objects Π, if we do not observe divergence of individual threads in the contextual refinement. That is, $\Pi \sqsubseteq^{\omega}_{\varphi} \Gamma$ is equivalent to linearizability and lock-freedom of Π, as shown in the following Theorem 6.2.

Theorem 6.2 (Additional Abstraction Theorem for LF)**.**

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \wedge \mathsf{LF}_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{\omega}_{\varphi} \Gamma$$

The contextual refinement $\sqsubseteq^{\omega}$ (see Definition 6.1) takes coarser observable behaviors than $\sqsubseteq^{\mathsf{t}\omega}$. It observes the divergence of the whole client program by using $\mathcal{O}_{\omega}$ at both the concrete and the abstract levels. Then the behaviors of Π correspond to those of Γ directly (i.e., we no longer need wrappers). Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract atomic object Γ.

For example, consider the client (6.3.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 1.1(a) and when it uses the abstract atomic one. The $\mathcal{O}_{\omega}$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_{\omega}$ set is $\{1::2::\epsilon, 2::1::\epsilon\}$ at both levels.

```
inc(); print(1);    ||    dec(); print(2);        (6.3.4)
```

Instead, if the client (6.3.4) uses the non-lock-free object in Figure 2.1(b), it may diverge and nothing is printed out (when the scheduling is unfair). The $\mathcal{O}_\omega$ set becomes $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq^{\omega}_{\varphi} \Gamma$ fails.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (6.3.5) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;        (6.3.5)
```

But for individual threads, divergence is not equivalent to non-termination, since a non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former case as divergence. For instance, in an unfair execution, the left thread of (6.3.5) may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (6.3.6),

```
while(true) skip;    ||    while(true) skip;        (6.3.6)
```

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

6.4 Abstraction for Starvation-Free and Deadlock-Free Objects

The contextual refinement for starvation-free and deadlock-free objects, $\Pi \sqsubseteq^{\mathsf{fair}}_{\varphi} \Pi'$, uses $\mathcal{O}_{\mathsf{fair}}$ at both the concrete and the abstract sides, ruling out undesired divergence caused by unfair scheduling. The abstract specifications Π' are generated by applying the wrappers $\mathsf{wr}_{\mathsf{SF}}$ and $\mathsf{wr}_{\mathsf{DF}}$ (in Figure 6.2) to the atomic specification Γ.

Wrapper for SF. Just like the wrapper $\mathsf{wr}_{\mathsf{WF}}$ for wait-freedom, the wrapper $\mathsf{wr}_{\mathsf{SF}}$ for starvation-freedom is an identity function. Under fair scheduling, a starvation-free object Π behaves just like an atomic object, both of which guarantee that every method call finishes.

As an example, consider the client program (6.4.1).

```
inc(); print(1);    ||    while(true) inc();        (6.4.1)
```

Intuitively, when the client uses the abstract atomic operation `INC`, any fair execution must print out 1. Thus $\mathcal{O}_{\mathsf{fair}}$ of the abstract program is a singleton set $\{1 :: \epsilon\}$. When the client uses the starvation-free object in Figure 2.1(d), its $\mathcal{O}_{\mathsf{fair}}$ set is still $\{1 :: \epsilon\}$. It does not produce more observable behaviors. But if it uses a non-starvation-free object (such as the deadlock-free one in Figure 2.1(b)), the left call to `inc` does not necessarily finish. The $\mathcal{O}_{\mathsf{fair}}$ set becomes $\{\epsilon, 1 :: \epsilon\}$. It produces more observable behaviors than the abstract client, so the contextual refinement fails.

Wrapper for DF. As the example (6.4.1) shows, $\sqsubseteq^{\mathsf{fair}}$ does not hold between a deadlock-free object Π and its atomic specification Γ. The reason is that deadlock-freedom does not guarantee thread progress, i.e., some method calls to Π may not finish, in fair executions. To re-establish the contextual refinement, we need the wrapper $\mathsf{wr}_{\mathsf{DF}}$ to delay the termination of the abstract atomic operations.

In $\mathsf{wr}_{\mathsf{DF}}(\langle C \rangle)$, we introduce a new object variable `done` (initialize to `false`), and use **while**-loops to allow the method to be delayed when other threads continuously finish the atomic block (in this case, `done` is infinitely often `true`). Also note `done` is reset to `false` at the end of the atomic block. Therefore, the **while**-loops eventually terminate in fair executions if other threads no longer sets `done` to `true`. This ensures whole program progress in fair executions, so DF holds.

Consider the example (6.4.1). When the client uses $\mathsf{wr}_{\mathsf{DF}}(\texttt{INC})$, the $\mathcal{O}_{\mathsf{fair}}$ set is $\{\epsilon, 1 :: \epsilon\}$. It is the same as the $\mathcal{O}_{\mathsf{fair}}$ set of the client using the deadlock-free counter.

As in $\mathsf{wr}_{\mathsf{LF}}$, we insert the **while**-loops both before and after the atomic operation. Thus a method is allowed to diverge either before or after its effects are visible to other threads.

6.5 Abstraction for PSF and PDF Objects

We define the wrapper in Figure 6.3 for each combination of progress (PSF or PDF) and fairness (`sfair` or `wfair`).

Wrapper for PSF under strongly fair scheduling. The wrapper $\mathsf{wr}_{\mathsf{PSF}}^{\mathsf{sfair}}$ is simply an identity function. It maps the atomic partial specification $\mathbf{await}(B)\{C\}$ to itself. This is because under *strongly fair scheduling* $\mathbf{await}(B)\{C\}$ will eventually be executed unless it is eventually always disabled. This is exactly what we need for PSF of linearizable objects, which requires that the invocation of each method eventually returns, unless the corresponding high-level atomic operation $\mathbf{await}(B)\{C\}$ is eventually always disabled (as specified by well-blocked in Figure 5.3).

Wrapper for PSF under weakly fair scheduling. Under weakly fair scheduling, however, we cannot guarantee that $\mathbf{await}(B)\{C\}$ is eventually executed even if B holds infinitely often. Therefore it alone cannot satisfy PSF. That's why we define $\mathsf{wr}_{\mathsf{PSF}}^{\mathsf{wfair}}(\mathbf{await}(B)\{C\})$, which guarantees that the atomic operation is eventually executed if B holds infinitely often. We introduce a blocking queue `listid` in the object state, which is a sequence of $(\mathsf{t}, \text{`}B\text{'})$ pairs, showing that the thread $\mathsf{t}$ requests to execute an atomic operation with the enabling condition $B$. Note that the enabling condition $B$ is recorded *syntactically* in `listid`, represented as '$B$'. The operator $\mathbf{enhd}(\texttt{listid})$ returns the first thread on the list whose enabling condition is true. It evaluates the syntactic enabling conditions '$B$' recorded in `listid` on the fly. Note that different pairs in `listid` may have different enabling conditions B. In the code generated by $\mathsf{wr}_{\mathsf{PSF}}^{\mathsf{wfair}}(\mathbf{await}(B)\{C\})$, we first append the current thread ID and the enabling condition 'B' at the end of the list. In the subsequent command the thread waits until both B holds and $\texttt{cid} = \mathbf{enhd}(\texttt{listid})$.[1] Then it atomically executes C and deletes the current thread in the queue.

This wrapper guarantees that C is eventually executed when B becomes infinitely often true because we know $B \wedge \texttt{cid} = \mathbf{enhd}(\texttt{listid})$ will be *eventually always* true, and then the weakly fair scheduling guarantees the execution of C. This is because, whenever B becomes true, either $\texttt{cid} = \mathbf{enhd}(\texttt{listid})$ holds or there is a pair (t', B') such that $B' \wedge \mathsf{t}' = \mathbf{enhd}(\texttt{listid})$ holds. In the first case, other threads trying to execute the object methods must be blocked at the **await**

[1] Actually the conjunct B in the **await** condition in the wrapper could be omitted, because B must be true when $\texttt{cid} = \mathbf{enhd}(\texttt{listid})$ holds.

command. Therefore B cannot be changed to false by other threads. Therefore $B \wedge \texttt{cid} = \textbf{enhd}(\texttt{listid})$ is always true until the current thread executes the atomic block. In the second case $\texttt{t}'$ must be able to finish its method, following the same argument above. Therefore there will be one less thread waiting in front of the current thread `cid`. Since B becomes true infinitely often, we eventually reach the first case.

As a result, the wrapper does not terminate in a weakly fair execution only if B is eventually always false. In that case the execution trace is well-blocked (see Figure 5.3), still satisfying PSF.

One may argue that the abstraction generated by the wrapper is not very useful because it may not be much simpler than the object implementation. For instance, if we consider the `acquire` method of locks, the abstraction is almost the same as queue locks or ticket locks. But we want to emphasize that our wrapper is a general one that works for any object method implementation with an atomic specification in the form of $\textbf{await}(B)\{C\}$. Therefore we know the method's progress-aware abstraction can always be in this form, no matter how complex its implementation is.

Wrapper for PDF under weakly fair scheduling. For the right column in Table 6.2, we first introduce the wrapper at the bottom right corner. The definition of PDF says a method can be non-terminating if (1) it is eventually always disabled, as specified by well-blocked (see Figure 5.3); or (2) there are always other method calls terminating, as specified by prog-p. Note that the second condition allows the method to be non-terminating even if it is eventually always enabled under weakly fair scheduling. As an example, the Treiber stack with a partial pop in Figure 6.4 demonstrates one such scenario. The `pop` method is blocked when the stack is empty. It is linearizable with respect to the following specification

$$\textbf{await}(\texttt{S} \neq \texttt{nil})\{\texttt{tmp} := \texttt{head}(\texttt{S}); \texttt{S} := \texttt{tail}(\texttt{S}); \}; \quad \textbf{return } \texttt{tmp}; \tag{6.5.1}$$

where `S` is the abstraction of the stack and `tmp` is a thread-local temporary variable.

```
initialize(){  Top := null; }

push(v){
 1  local x, b, t;
 2  b := false;
 3  x := cons(v, null);
 4  while (!b) {
 5    t := Top;
 6    x.next := t;
 7    b := cas(&Top, t, x);
 8  }
}
```

```
pop(){
 9  local x, b, t, v;
10  b := false;
11  while (!b) {
12    t := Top;
13    if (t != null) {
14      v := t.data;
15      x := t.next;
16      b := cas(&Top, t, x);
17    }
18  }
19  return v;
}
```

```
initialize'(){  initialize(); done := false;}
push'(v){ push(v); DLY_NOOP}
pop'(v){ tmp := pop(); DLY_NOOP; return tmp}
DLY_NOOP ≝ await(¬done){done := true}; done := false;
```

```
r0 := pop'();
print(r0);
```

||

```
push'(1);
push'(2);
r1 := pop'();
print(r1);
while(true){ push'(0) };
```

Figure 6.4: Treiber stacks with partial pops.

In the following execution context (6.5.2),

```
pop();   ||   while(true){ push(0); }          (6.5.2)
```

the call of the concrete method `pop` may never terminate because its **cas** command may always fail, although the enabling condition at the abstract level (`S` $\neq$ `nil`) is eventually always true. However, if we replace the method implementation with the specification (6.5.1), `pop` must terminate under weakly fair scheduling. This shows that the concrete implementation cannot contextually refine this simple specification (6.5.1).

Our first attempt to address this problem is to introduce a new object variable `done` (initialized to `false`), and let the wrapper $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}$ transform **await**$(B)\{C\}$ into:

$$\textbf{await}(B \wedge \neg\texttt{done})\{C;\ \texttt{done} := \texttt{true}\};\quad \texttt{done} := \texttt{false}; \tag{6.5.3}$$

Therefore the resulting **await** command may not be executed even if B is always true, because `done` can be set to `true` infinitely often when other threads finish the atomic block. Also note `done` is reset to `false` at the end of each **await** command, therefore the condition $\neg$`done` cannot always disable the **await** command, which may cause deadlock. As a result, there is always some thread that can finish the wrapper (i.e., prog-p holds) unless the B-s of all the pending invocations are eventually always false (i.e., well-blocked holds), thus PDF holds.

However, this is not the end of the story. If the code (6.5.3) fails to terminate, C must not be executed and no effects (over the object data) are generated. However, it is possible for PDF methods to finish C and make the effects visible to other threads but fail to terminate. As an example we define the `push'` and `pop'` methods in Figure 6.4 as a new implementation of the Treiber stack. They call the `push` and `pop` methods respectively and then execute the code snippet `DLY_NOOP` before they return. `DLY_NOOP` simply waits until `done` becomes `false` and then atomically sets it to `true`, and finally resets it to `false`. The only purpose of `DLY_NOOP` is to allow the methods to be delayed by other threads or to delay others.

Then we consider the client code shown at the bottom of Figure 6.4. Under weakly fair scheduling it is possible that the call of `pop'()` by the left thread never terminates but the thread on the right prints out 1. That is, although the `pop'()` on the left does not terminate, it does generate effects over the stack and the effects happen before the `pop'()` on the right. Such an external event trace cannot be generated if we replace the concrete `push'()` and `pop'()` methods with the abstract method code generated using the wrapper (6.5.3) defined above. Thus the contextual refinement between the concrete code and the wrapped specification does not hold.

Our solution is to append an **await** command at the end of (6.5.3), so that the resulting code $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ (see Figure 6.3) may finish C but still be blocked at the end.

Wrapper for PDF under strongly fair scheduling. Much of the effort to define $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ is to allow the resulting code to be non-terminating even if B is eventually always true. We need to do the same to define $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$, but it is more challenging with *strongly fair scheduling* because $\mathbf{await}(\neg\texttt{done})\{\}$ cannot be blocked under strongly fair scheduling if `done` is infinitely often true. Therefore we use **while**-loops in $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ to allow the method to be delayed when `done` is infinitely often true.[2] Note that **while** (`done`){} terminates when `done` is false. We can see the similarity between $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}$ and $\mathsf{wr}_{\mathsf{DF}}$ (defined in Figure 6.2).

Wrappers for the state abstraction function. Since the program transformations by the wrappers introduce new object variables such as `listid` and `done`, we need to change the state abstraction function φ accordingly, which is defined as $\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\varphi)$ in Figure 6.3 ($\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$ and $\mathsf{Prog} \in \{\mathsf{PSF}, \mathsf{PDF}\}$).

More discussions. There could be different ways to define the wrappers to validate the Abstraction Theorem 6.1. We do not intend to claim that our definitions are the simplest ones (and it is unclear how to formally compare the complexity of different wrappers), but we would like to point out that, although some of the wrappers look complex, the complexity is partly due to the effort to have general wrappers that work for any atomic specifications in the form of $\mathbf{await}(B)\{C\}$. It is possible to have simpler wrappers for specific objects. For instance, the lock specification Γ in (2.3.2) defined in Subsection 2.3 can already serve

[2] Actually the conjunct $\neg$`done` in the **await** condition in the wrapper could be removed, because the loop **while** (`done`){} before the **await** block can already produce the non-terminating behaviors when other threads finish the method infinitely often (i.e., `done` is infinitely often true). Here we keep the conjunct $\neg$`done` to make the wrapper more intuitive.

as an abstraction for the test-and-set lock object Π_{TAS} (which is a PDF lock) under weakly fair scheduling, i.e., $\Pi_{\text{TAS}} \sqsubseteq_{\varphi}^{\mathsf{wfair}} \Gamma$ holds.

7

Verifying Progress of Concurrent Objects

In this section, we explain the program logic LiLi. It is a rely-guarantee style program logic to verify linearizability and all the six progress properties. It also establishes progress-aware contextual refinements between concrete object implementations and abstract specifications.

We first analyze the challenges and explain our approach informally. Then we present the assertion language, the inference rules and the logic soundness theorem. In particular, we focus on the definite actions and the delaying actions that we introduced to support blocking and delay respectively. We start with the rules to reason about starvation-free and deadlock-free objects, which can be degenerated to reason about wait-free and lock-free objects. Then we generalize the rules to also support partial methods. Along with the explanations of the rules, we show some small examples. Finally we discuss the soundness theorem and show more applications of LiLi.

7.1 Challenges and Key Ideas

In Subsection 2.4 we have given an overview of the traditional rely-guarantee logic for safety proofs (Jones, 1983), and the way to encode linearizability verification in the logic. We have also explained the challenges in supporting progress verification, as outlined below:

- Non-termination caused by interference. There are two different kinds of interference that may cause thread non-termination, namely *blocking* and *delay*. In some algorithms (e.g., the optimistic list), blocking and delay can be intertwined, makes the reasoning significantly challenging.

- Avoid circular reasoning. Rely-guarantee-style logics relies on circular reasoning to support thread-modular verification, but circular reasoning is usually unsound in liveness verification. Both blocking and delay may cause circular dependency of progress. We need to distinguish "good" blocking and delay from "bad" ones.

- Ad-hoc synchronization and dynamic locks. Enforcing lock orders is a natural way to avoid circularity, but it is sometimes difficult to identify certain object fields as locks. Also, many objects with explicit lock fields are dynamic (e.g., the lock-coupling list), so the lock orders need to be determined dynamically, making the verification challenging.

To address these problems, our logic enforces the following principles to permit restricted forms of blocking and delay, but prevent circular reasoning and non-termination.

First, if a thread is blocked, the events it waits for must eventually occur. To avoid circular reasoning, we find "definite actions" of each thread, which under fair scheduling will definitely happen once enabled, regardless of the interference from the environment. Then each blocked thread needs to show it waits for only a finite number of definite actions from the environment threads. They form an acyclic queue, and there is always at least one of them enabled. This is what we call "definite progress", which is crucial for proving starvation-freedom.

Second, actions of a thread can delay others *only if* they are making the executing object method to move towards termination. Each object method can only execute a finite number of such delaying actions to avoid indefinite delay. This is enforced by assigning a finite number of tokens to each method. A token must be paid to execute a delaying action.

Third, we divide actions of a thread into normal ones (which do not delay others) and delaying ones, and further stratify delaying actions into multiple levels. When a thread is delayed by a level-k action from its environment, it is allowed to execute not only more normal actions, but also more delaying actions at lower levels. Allowing one delaying action to trigger more steps of other delaying actions is necessary for verifying algorithms with nested locks and rollbacks, such as the optimistic lists in Figure 2.3. The stratification prevents the circular delay in the example of Figure 2.4.

Fourth, our delaying actions and definite actions are all semantically specified as part of object specifications, therefore we can support ad-hoc synchronization and do not rely on built-in synchronization primitives to enforce ordering of events. Moreover, since the specifications are all parametrized over states, they are expressive enough to support dynamic locks as in lock-coupling lists. Also our "definite progress" condition allows each blocked thread to decide *locally* and *dynamically* a queue of definite actions it waits for. There is no need to enforce a global ordering of blocking dependencies agreed by every thread. This also provides thread-modular support of dynamic locks.

Below we give more details about some of these key ideas.

7.1.1 Using Tokens to Prevent Infinite Loops

The key to ensuring termination is to require each loop to terminate. Earlier work (Hoffmann *et al.*, 2013; Liang *et al.*, 2014) requires each round of the loop to consume resources called tokens. The rule for loops is in the following form:

$$\frac{P \wedge B \Rightarrow P' * \Diamond \qquad R, G \vdash \{P'\}C\{P\}}{R, G \vdash \{P\}\textbf{while}\ (B)\ C\{P \wedge \neg B\}}\ (\textsc{term})$$

Here $\Diamond$ represents one token, and "$*$" is the normal separating conjunction in separation logic. The premise says the precondition P' of the loop body C has one less token than P, showing that one token needs to be consumed to start this new round of loop. Since the number of tokens strictly decreases, we know the loop must terminate when the thread has no token.

We use this simple idea to enforce termination of loops, and extend it to handle blocking and delay in a concurrent setting.

7.1.2 Using Tokens to Prevent Indefinite Delays

To support delay, we first identify the delaying actions in each method. We explicitly specify in the rely/guarantee conditions which steps could delay the progress of other threads.

To prohibit unlimited delays without making progress, we assign a finite number m of ⧫-tokens to an object method, and require that a thread can do at most m delaying actions before the method finishes. Whenever a step of thread $\mathtt{t}'$ delays the progress of thread $\mathtt{t}$, we require $\mathtt{t}'$ to consume one ⧫-token. At the same time, thread $\mathtt{t}$ could increase its ◊-tokens (see the TERM rule above) so that it can loop more rounds. In a sense the thread $\mathtt{t}'$ transfers its ⧫-token to thread $\mathtt{t}$ which is then converted to one or more ◊-tokens upon receipt. Similar token transfer ideas have been used to verify lock-free algorithms in earlier work (Hoffmann *et al.*, 2013; Liang *et al.*, 2014).

7.1.3 Definite Actions and Definite Progress

Our approach to cut the blocking-caused circular dependency is inspired by the implementation of ticket locks, which is used to implement the starvation-free counter `inc_tkL` in Figure 2.1(d). We can see that `inc_tkL` is not concerned with the circular dependency problem. Intuitively the ticket lock algorithm in Figure 2.1(c) ensures that the threads requesting the lock always constitute a queue $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n$. The head thread, $\mathtt{t}_1$, gets the ticket number which equals `owner` and can immediately acquire the lock. Once it releases the lock (by increasing `owner`), $\mathtt{t}_1$ is dequeued. Moreover, for any thread $\mathtt{t}$ in this queue, the number of threads ahead of $\mathtt{t}$ never increases. Thus $\mathtt{t}$ must eventually become the head of the queue and acquire the lock. Here the dependencies among progress of the threads are in concert with the queue.

Following this queue principle, we explicitly specify the queue of progress dependencies in our logic to avoid circular reasoning.

Definite actions. First, we introduce a novel notion called a "definite action" $\mathcal{D}$, which models a thread action that, once enabled, must be eventually finished regardless of what the environment does. In detail, $\mathcal{D}$ is in the form of $P_d \rightsquigarrow Q_d$. It requires in every execution that Q_d should eventually hold *if* P_d holds, and P_d should be preserved (by both the current thread and the environment) until Q_d holds. For `inc_tkL`, the definite action $P_d \rightsquigarrow Q_d$ of a thread can be defined as follows. P_d says that `owner` equals the thread's ticket number `i`, and Q_d says that `owner` has been increased to $\texttt{i} + 1$. That is, a thread definitely releases the lock when acquiring it. Of course we have to ensure in our logic that $\mathcal{D}$ is indeed definite. We will explain in detail the logic rule that enforces it in Subsection 7.2.2.

Definite progress. Then we use definite actions to prove termination of loops. We need to first find an assertion Q specifying the condition when the thread `t` can progress on its own, i.e., it is *not* blocked. Then we enforce the following principles:

1. If Q is continuously true, we need to prove the loop terminates following the idea of the TERM rule;
2. If Q is false, the following must *always* be true:
   (a) There is a finite queue of definite actions of other threads that the thread `t` is waiting for, among which there is at least one (from a certain thread $\texttt{t}'$) enabled. The length of the queue is E.
   (b) E decreases whenever one of these definite actions is finished;
   (c) The expression E is never increased by any threads (no matter whether Q holds or not); and it is non-negative.

We can see E serves as a well-founded metric. By induction over E we know eventually Q holds, which implies the termination of the loop by the above condition 1.

These conditions are enforced in our new inference rule for loops, which extends the TERM rule (in Subsection 7.1.1) and is presented in

Subsection 7.2.2. The condition 2 shows the use of definite actions in our reasoning about progress. We call it the "definite progress" condition.

The reasoning above implicitly makes use of the fairness assumption. The fair scheduling ensures that the environment thread $\mathtt{t}'$ mentioned in the condition 2(a) is scheduled infinitely often, therefore its definite action will definitely happen. By conditions 2(b) and 2(c) we know E will become smaller. In this way E keeps decreasing until Q holds eventually.

For `inc_tkL`, Q is defined as $(\mathtt{i} = \mathtt{owner})$ and the metric E is $(\mathtt{i} - \mathtt{owner})$. Whenever an environment thread $\mathtt{t}'$ finishes a definite action by releasing the lock, it increases `owner`, so E decreases. When E is decreased to 0, the current thread is unblocked. Its loop terminates and it succeeds in acquiring the lock.

7.1.4 Allowing Queue Jumps for Deadlock-Free Objects

The method `inc` in Figure 2.1(b) implements a deadlock-free counter using the TAS lock. If the current thread `t` waits for the lock, we know the queue of definite actions it waits for is of length *one* because it is possible for the thread to acquire the lock immediately after the lock is released. However, another thread $\mathtt{t}'$ may preempt `t` and do a successful **cas**. Then thread `t` is blocked and waits for a queue of definite actions again. This delay caused by thread $\mathtt{t}'$ can be viewed as a queue jump in our definite-progress-based reasoning. Actually `inc` cannot satisfy the definite progress requirement because we cannot find a strictly decreasing queue size E. It is not starvation-free.

However, the queue jump here is acceptable when verifying deadlock-freedom. This is because thread $\mathtt{t}'$ delays `t` only if $\mathtt{t}'$ successfully acquires the lock, which allows it to eventually finish the `inc` method. Thus the system as a whole progresses.

Nevertheless, as explained in Subsection 2.4.3, we have to make sure the queue jump (which is a special form of delay) is a "good" one.

The token-transfer ideas we explained in Subsection 7.1.2 can be used to support disciplined queue jumps. As in Subsection 7.1.2, we assign a finite number m of ⧫-tokens to an object method. When a step of thread $\mathtt{t}'$ delays the progress of thread `t` (including jumping

its queue), thread $\mathtt{t}'$ should consume one ⧫-token, but thread $\mathtt{t}$ could increase its ◊-tokens to loop more rounds. Similarly, we also redefine the definite progress condition to allow the metric E (about the length of the queue) to be increased when an environment thread jumps the queue at the cost of a ⧫-token.

7.1.5 Allowing Rollbacks for Optimistic Locking

The ideas we just explained already support simple deadlock-free objects such as `inc` in Figure 2.1(b), but they cannot be applied to objects with optimistic synchronization, such as optimistic lists (Herlihy and Shavit, 2008) and lazy lists (Heller *et al.*, 2005).

Figure 2.3 shows part of the optimistic list implementation, which we have explained in Subsection 2.4.3. For this object, when the validation fails, a thread will release the locks it has acquired and roll back. Thus the thread may acquire the locks for an unbounded number of times. Since each lock acquirement will delay other threads requesting the same lock and each delaying action should consume one ⧫-token, it seems that the thread would need an infinite number of ⧫-tokens, which we prohibit in the preceding subsection to ensure deadlock-freedom, even though this list object is indeed deadlock-free.

We generalize the token-transfer ideas to allow rollbacks in order to verify this kind of optimistic algorithms, but still have to be careful to avoid the circular delay caused by the "bad" rollbacks in Figure 2.4, as we explain in Subsection 2.4.3.

Our solution is to stratify the delaying actions. Each action is now labeled with a level k. The normal actions which cannot delay other threads are at the lowest level 0. The ⧫-tokens are stratified accordingly. A thread can roll back and do more actions at level k only when its environment does an action at a higher level k', at the cost of a k'-level ⧫-token. Note that the ⧫-tokens at the highest level are strictly decreasing, which means a thread cannot roll back its actions of the highest level. In fact, the numbers of ⧫-tokens at all levels constitute a tuple $(m_k, \ldots, m_2, m_1)$. It is strictly descending along the dictionary order.

The stratified ⧫-tokens naturally prohibit the circular delay problem discussed in Subsection 2.4.3 with the object in Figure 2.4. The deadlock-free optimistic list in Figure 2.3 can now be verified. We classify its delaying actions into two levels. Actions at level 2 (the highest level) update the list, which correspond to line 9 in Figure 2.3, and each method can do only *one* such action. Lock acquirements are classified at level 1, so a thread is allowed to roll back and re-acquire the locks when its environment updates the list.

7.2 The Program Logic LiLi

LiLi verifies the linearizability of objects by proving the method implementations refine abstract atomic operations. The top level judgment is in the form of $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$. (The OBJ rule for this judgment is given in Figure 7.4 and will be explained later.) To verify an object Π, we give a corresponding object specification Γ (see Figure 3.1), which maps method names to atomic commands. In addition, we need to provide an object invariant (P) and rely/guarantee conditions (R and G) for the refinement reasoning in a concurrent setting. Here P is a relational assertion that specifies the consistency relation between the concrete data representation and the abstract value. Similarly, R and G lift regular rely and guarantee conditions to the binary setting, which now specify transitions of states at both the concrete level and the abstract level. The definite actions $\mathcal{D}$ is a special form of state transitions used for *progress* reasoning. The definitions of P, R, G and $\mathcal{D}$ are shown in Subsection 7.2.1.

Note LiLi is a logic for concurrent objects Π only. We do not provide logic rules for clients. See Subsection 7.3 for more discussions.

To simplify the presentation in this tutorial, we describe LiLi based on the plain Rely-Guarantee Logic (Jones, 1983). Also, to avoid "variables as resources" (Parkinson *et al.*, 2006), we assume program variables are either thread-local or read-only. The full version of LiLi (Liang and Feng, 2018b) extends the more advanced Rely-Guarantee-based logic LRG (Feng, 2009) to support dynamic allocation and ownership transfer. It also drops the assumption about program variables.

$$
\begin{array}{rrcl}
(\mathit{RelAssn}) & P, Q, J & ::= & B \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \\
& & & \mid [\![p]\!] \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \ldots \\
(\mathit{RelAct}) & R, G & ::= & P \ltimes_k Q \mid [P] \mid \mathcal{D} \\
& & & \mid \lfloor G \rfloor_0 \mid G \wedge G \mid G \vee G \mid \ldots \\
(\mathit{DAct}) & \mathcal{D} & ::= & P \rightsquigarrow Q \mid \forall x. \mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\
(\mathit{FullAssn}) & p, q & ::= & P \mid \mathsf{arem}(C) \mid \Diamond(E) \mid \blacklozenge(E_k, \ldots, E_1) \\
& & & \mid \lfloor p \rfloor_{\mathsf{a}} \mid \lfloor p \rfloor_{\Diamond} \mid p * q \mid p \wedge q \mid p \vee q \mid \ldots
\end{array}
$$

Figure 7.1: Syntax of the assertion language.

7.2.1 Assertions

We define assertions in Figure 7.1. The relational state assertions P and Q specify the relationship between the concrete state σ and the abstract state Σ. Here we use $\mathbb{s}$ and $\mathbb{h}$ for the store and the heap at the abstract level (see Figure 3.2). For simplicity, we assume the program variables used in the concrete code are different from those in the abstract code (e.g., we use `x` and `X` at the concrete and abstract levels respectively). We use the relational state $\mathfrak{S}$ to represent the pair of states (σ, Σ), as defined in Figure 7.2.

Figure 7.2(a) defines semantics of state assertions. The boolean expression B holds if it evaluates to true at the combined store of s and $\mathbb{s}$. emp describes empty heaps. The assertion $E_1 \mapsto E_2$ specifies a singleton heap at the concrete level with the value of the expression E_2 stored at the location E_1. Its counterpart for an abstract level heap is represented as $E_1 \Mapsto E_2$. Semantics of separating conjunction $P * Q$ is similar as in separation logic, except that it is now lifted to relational states $\mathfrak{S}$. The disjoint union of two relational states is defined at the top of the figure. Semantics of the assertion $[\![p]\!]$ will be defined latter (see Figure 7.2(c)).

Rely/guarantee assertions R and G specify the transitions over the relational states $\mathfrak{S}$. Their semantics is defined in Figure 7.2(b). The action $P \ltimes_k Q$ says that the initial relational states satisfy P and the resulting states satisfy Q. We can ignore the index k for now, which is used to stratify actions that may delay the progress of other threads and will be explained in Subsection 7.2.3. $[P]$ specifies identity

$$\mathfrak{S} ::= (\sigma, \Sigma) \qquad (\sigma,\Sigma) \uplus (\sigma',\Sigma') \stackrel{\text{def}}{=} (\sigma \uplus \sigma', \Sigma \uplus \Sigma')$$
$$\text{where} \quad (s,h) \uplus (s',h') \stackrel{\text{def}}{=} (s, h \uplus h') \ , \quad \text{if } s = s'$$

$$\begin{array}{lll}
((s,h),(\mathbb{s},\mathbb{h})) \models B & \text{iff} & [\![B]\!]_{s \uplus \mathbb{s}} = \mathbf{true} \\
((s,h),(\mathbb{s},\mathbb{h})) \models \mathsf{emp} & \text{iff} & dom(h) = dom(\mathbb{h}) = \emptyset \\
((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \mapsto E_2 & \text{iff} & h = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \rightsquigarrow [\![E_2]\!]_{s \uplus \mathbb{s}}\} \\
((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \Mapsto E_2 & \text{iff} & \mathbb{h} = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \rightsquigarrow [\![E_2]\!]_{s \uplus \mathbb{s}}\} \\
\mathfrak{S} \models P * Q & \text{iff} & \exists \mathfrak{S}_1, \mathfrak{S}_2.\, \mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2 \\
 & & \quad \wedge (\mathfrak{S}_1 \models P) \wedge (\mathfrak{S}_2 \models Q)
\end{array}$$

(a) Semantics of relational state assertions P and Q.

$$\begin{array}{lll}
(\mathfrak{S},\mathfrak{S}') \models P \ltimes_{k'} Q & \text{iff} & (\mathfrak{S} \models P) \wedge (\mathfrak{S}' \models Q) \\
(\mathfrak{S},\mathfrak{S}') \models [P] & \text{iff} & (\mathfrak{S}' = \mathfrak{S}) \wedge (\mathfrak{S} \models P)
\end{array}$$

(b) Semantics of relational rely/guarantee assertions R and G.

$$\begin{array}{lll}
(\mathfrak{S},(u,w),C) \models P & \text{iff} & \mathfrak{S} \models P \\
(\mathfrak{S},(u,w),C) \models \mathsf{arem}(C') & \text{iff} & C = C' \\
(\mathfrak{S},(u,w),C) \models \Diamond(E) & \text{iff} & \exists n.\, ([\![E]\!]_{\mathfrak{S}.s} = n) \wedge (n \leq w) \\
(\mathfrak{S},(u,w),C) \models \blacklozenge(E_k,\ldots,E_1) & \text{iff} & ([\![E_k]\!]_{\mathfrak{S}.s},\ldots,[\![E_1]\!]_{\mathfrak{S}.s}) \leq u \\
(\mathfrak{S},(u,w),C) \models \lfloor p \rfloor_{\Diamond} & \text{iff} & \exists w'.\, (\mathfrak{S},(u,w'),C) \models p \\
(\mathfrak{S},(u,w),C) \models \lfloor p \rfloor_{\mathsf{a}} & \text{iff} & \exists C'.\, (\mathfrak{S},(u,w),C') \models p \\
\mathfrak{S} \models [\![p]\!] & \text{iff} & \exists u,w,C.\, (\mathfrak{S},(u,w),C) \models p
\end{array}$$
$$C \uplus C' \stackrel{\text{def}}{=} \begin{cases} C' & \text{if } C = \mathbf{skip} \\ C & \text{if } C' = \mathbf{skip} \end{cases}$$
$$(\mathfrak{S},(u,w),C) \uplus (\mathfrak{S}',(u',w'),C') \stackrel{\text{def}}{=} (\mathfrak{S} \uplus \mathfrak{S}', (u+u', w+w'), C \uplus C')$$

(c) Semantics of full assertions p and q.

Figure 7.2: Semantics of assertions.

transitions with the initial states satisfying P. Semantics of $\lfloor G \rfloor_0$ is defined in Subsection 7.2.3 too (see Figure 7.8). Below we use $P \ltimes Q$ as a shorthand for $P \ltimes_0 Q$. We also use Id for $[\mathsf{true}]$, which represents arbitrary identity transitions.

We further instrument the relational state assertions with the specifications of the abstract level code and various tokens. The resulting *full assertions* p and q are defined in Figure 7.1, whose semantics is given in Figure 7.2(c). The assertion p is interpreted over $(\mathfrak{S}, (u, w), C)$. C is the abstract-level code that remains to be refined. It is specified by the assertion $\mathsf{arem}(C)$. Since our logic verifies linearizability of objects, C is always in the form of atomic commands $\langle C' \rangle$ (ahead of **return** commands). The pair (u, w) records the numbers of various tokens $\blacklozenge$ and $\lozenge$. It serves as a well-founded metric for our progress reasoning. Informally w specifies the upper bound of the round of loops that the current thread can execute if it is neither blocked nor delayed by its environment. The assertion $\lozenge(E)$ says the number w of $\lozenge$-tokens is *no less than* E. Therefore $\lozenge(0)$ always holds, and $\lozenge(n+1)$ implies $\lozenge(n)$ for any n. We postpone the explanation of u and the assertion $\blacklozenge(E_k, \ldots, E_1)$ to Subsection 7.2.3. Below we use $\lozenge$ as the shorthand for $\lozenge(1)$. We use $\lfloor p \rfloor_\lozenge$ to ignore the descriptions in p about the number of tokens. $[\![p]\!]$ converts p back to a relational state assertion.

Separating conjunction $p * q$ has the standard meaning as in separation logic, which says p and q hold over disjoint parts of $(\mathfrak{S}, (u, w), C)$ respectively (the formal definition elided here). The disjoint union is defined in Figure 7.2(c). The disjoint union of the numbers of tokens is the sum of them. The disjoint union of C_1 and C_2 is defined only if at least one of them is **skip**. Therefore we know the following holds (for any P and C):

$$(P \wedge \mathsf{arem}(C) \wedge \lozenge) * (\lozenge \wedge \mathsf{emp}) \;\Leftrightarrow\; (P \wedge \mathsf{arem}(C) \wedge \lozenge(2))$$

Definite actions. Figure 7.1 also defines definite actions $\mathcal{D}$, whose semantics is given in Figure 7.3(a). $P \rightsquigarrow Q$ specifies the transitions where the final states satisfy Q *if* the initial states satisfy P. It is different from $P \ltimes Q$ in that $P \rightsquigarrow Q$ does not restrict the transitions if initially P does not hold. Consider the following example $\mathcal{D}_x$.

$$\mathcal{D}_x \;\stackrel{\text{def}}{=}\; \forall n.\ ((\mathtt{x} \mapsto n) \wedge (n > 0)) \rightsquigarrow (\mathtt{x} \mapsto n + 1)$$

$\mathcal{D}_x$ describes the state transitions which increment $\mathtt{x}$ if $\mathtt{x}$ is positive initially. It is satisfied by any transitions where initially $\mathtt{x}$ is not positive.

$$
\begin{array}{ll}
(\mathfrak{S}, \mathfrak{S}') \models P \rightsquigarrow Q & \text{iff } (\mathfrak{S} \models P) \implies (\mathfrak{S}' \models Q) \\
(\mathfrak{S}, \mathfrak{S}') \models \forall x.\, \mathcal{D} & \text{iff } \forall n.\, (\mathfrak{S}\{x \rightsquigarrow n\}, \mathfrak{S}'\{x \rightsquigarrow n\}) \models \mathcal{D} \\
(\mathfrak{S}, \mathfrak{S}') \models \mathcal{D}_1 \wedge \mathcal{D}_2 & \text{iff } ((\mathfrak{S}, \mathfrak{S}') \models \mathcal{D}_1) \wedge ((\mathfrak{S}, \mathfrak{S}') \models \mathcal{D}_2)
\end{array}
$$

(a) Semantics of $\mathcal{D}$.

$$
\begin{array}{rcl}
\mathsf{Enabled}(P \rightsquigarrow Q) & \stackrel{\text{def}}{=} & P \\
\mathsf{Enabled}(\forall x.\, \mathcal{D}) & \stackrel{\text{def}}{=} & \exists x.\, \mathsf{Enabled}(\mathcal{D}) \\
\mathsf{Enabled}(\mathcal{D}_1 \wedge \mathcal{D}_2) & \stackrel{\text{def}}{=} & \mathsf{Enabled}(\mathcal{D}_1) \vee \mathsf{Enabled}(\mathcal{D}_2) \\
\langle \mathcal{D} \rangle & \stackrel{\text{def}}{=} & \mathcal{D} \wedge (\mathsf{Enabled}(\mathcal{D}) \ltimes \mathsf{true}) \\
[\mathcal{D}] & \stackrel{\text{def}}{=} & \mathsf{Enabled}(\mathcal{D}) \rightsquigarrow \mathsf{Enabled}(\mathcal{D})
\end{array}
$$

$$
\mathcal{D}' \leqslant \mathcal{D} \quad \text{iff} \quad (\mathsf{Enabled}(\mathcal{D}') \Rightarrow \mathsf{Enabled}(\mathcal{D})) \wedge (\mathcal{D} \Rightarrow \mathcal{D}')
$$

(b) Useful syntactic sugars.

Figure 7.3: Semantics of definite actions.

The conjunction $\mathcal{D}_1 \wedge \mathcal{D}_2$ is useful for enumerating definite actions. For instance, when the program uses two locks `L1` and `L2`, the definite action $\mathcal{D}$ of the whole program is usually in the form of $\mathcal{D}_1 \wedge \mathcal{D}_2$, where $\mathcal{D}_1$ and $\mathcal{D}_2$ specify `L1` and `L2` respectively.

We define some useful syntactic sugars in Figure 7.3(b). The state assertion $\mathsf{Enabled}(\mathcal{D})$ takes the guard condition of $\mathcal{D}$. We use $\langle \mathcal{D} \rangle$ to represent the state transitions of $\mathcal{D}$ when it is enabled at the initial state. Intuitively $\langle \mathcal{D} \rangle$ gives us the corresponding "$\ltimes$" actions. For instance, $\langle P \rightsquigarrow Q \rangle$ is equivalent to $P \ltimes Q$. For the example $\mathcal{D}_x$ defined above, $\langle \mathcal{D}_x \rangle$ is equivalent to the following:

$$
\exists n.\, ((\mathtt{x} \mapsto n) \wedge (n > 0)) \ltimes (\mathtt{x} \mapsto n + 1)
$$

In addition, we define the syntactic sugar $[\mathcal{D}]$ as a definite action describing the preservation of $\mathsf{Enabled}(\mathcal{D})$. For the example $\mathcal{D}_x$ above, $[\mathcal{D}_x]$ represents the following definite action:

$$
(\exists n.\, (\mathtt{x} \mapsto n) \wedge (n > 0)) \rightsquigarrow (\exists n.\, (\mathtt{x} \mapsto n) \wedge (n > 0))
$$

It specifies the transitions which keep `x` positive if it is positive initially. The notation $\mathcal{D}' \leqslant \mathcal{D}$ will be explained later in Subsection 7.2.2. Since

$\mathcal{D}$ is a special rely/guarantee assertion, the semantics of $\mathcal{D} \Rightarrow \mathcal{D}'$ follows the standard meaning of $R \Rightarrow R'$ (Feng, 2009) (or see the definition in Figure 7.8).

Thread IDs as implicit assertion parameters. All the assertions in our logic, including state assertions, rely/guarantee conditions and definite actions, are implicitly parametrized over thread IDs. Although our logic does modular reasoning about the object code without any knowledge about clients, it is useful for assertions to refer to thread IDs. For instance, we may use $\mathtt{L} \mapsto \mathtt{t}$ to represent that the lock $\mathtt{L}$ is acquired by the thread $\mathtt{t}$. We use $P_{\mathtt{t}}$ to represent the instantiation of the thread ID parameter in P with $\mathtt{t}$, which means P holds on thread $\mathtt{t}$. Then P alone can also be understood as $\forall \mathtt{t}.P_{\mathtt{t}}$, and $P \Rightarrow Q$ can be viewed as $\forall \mathtt{t}.P_{\mathtt{t}} \Rightarrow Q_{\mathtt{t}}$. The same convention applies to rely/guarantee conditions and definite actions.

7.2.2 Verifying Starvation-Freedom with Definite Actions

Figures 7.4, 7.5 and 7.14 present the inference rules of LiLi. We explain the logic in three steps. In this subsection we explain the use of definite actions to reason about starvation-freedom. Then we explain the delay mechanism for deadlock-freedom in Subsection 7.2.3. Finally we explain the supports for partial methods in Subsection 7.2.4.

The OBJ *Rule*

As the top rule of the logic, the OBJ rule says that, to verify Π satisfies its specification Γ with the object invariant P, one needs to specify the rely/guarantee conditions R and G, and the definite actions $\mathcal{D}$, and then prove that each individual object method implementation refines its specification. In general Γ must be an *atomic partial specification*. For objects with total methods, Γ must be total, written as $\mathsf{total}(\Gamma)$.

For each method, we take the object invariant P and the annotated preconditions $\mathcal{P}$ (in Π) as preconditions. The object invariant P should ensure that the annotated pre-conditions $\mathcal{P}$ and $\mathcal{P}'$ (in Γ) are either both true or both false. That is, whenever P holds, it is either safe to

$$\frac{\begin{array}{c}\text{for all } f \in dom(\Pi):\quad \Pi(f) = (\mathcal{P}, x, C) \qquad \Gamma(f) = (\mathcal{P}', y, C') \\ P \Rightarrow (\mathcal{P} \wedge \mathcal{P}') \vee (\neg\mathcal{P} \wedge \neg\mathcal{P}') \\ \mathcal{D}, R, G \vdash \{P \wedge \mathcal{P} \wedge (x = y) \wedge \mathsf{arem}(C') \wedge \blacklozenge(\vec{E})\}\, C\, \{P \wedge \mathsf{arem}(\mathbf{skip})\} \\ \forall \mathsf{t}, \mathsf{t}'.\ \mathsf{t} \neq \mathsf{t}' \Longrightarrow G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'} \qquad \mathsf{wffAct}(R, \mathcal{D}) \qquad P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})\end{array}}{\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma}\ (\textsc{obj})$$

$$\frac{\begin{array}{c}p \wedge B \Rightarrow p' \qquad p \wedge B \wedge (\mathsf{Enabled}(\mathcal{D}) \vee Q) \Rightarrow p' * (\Diamond \wedge \mathsf{emp}) \\ \mathcal{D}, R, G \vdash \{p'\} C \{p\} \qquad p \wedge B \Rightarrow J \wedge \mathsf{arem}(\mathbf{await}(B')\{C'\}) \\ \mathsf{Sta}(J, R \vee G) \quad J \Rightarrow (R, G \colon \mathcal{D}' \xrightarrow{f} (Q, B')) \quad \mathcal{D}' \leqslant \mathcal{D} \quad \mathsf{wffAct}(R, \mathcal{D}')\end{array}}{\mathcal{D}, R, G \vdash \{p\}\mathbf{while}\ (B)\{C\}\{p \wedge \neg B\}}\ (\textsc{whl})$$

$$\frac{\vdash [p] C [q'] \qquad q' \Rrightarrow_k q \qquad (\llbracket p \rrbracket \ltimes_k \llbracket q \rrbracket) \Rightarrow G}{\mathcal{D}, \mathsf{Id}, G \vdash \{p\}\langle C \rangle\{q\}}\ (\textsc{atom})$$

$$\frac{\mathcal{D}, \mathsf{Id}, G \vdash \{p\}\langle C \rangle\{q\} \qquad \mathsf{Sta}(\{p, q\}, R)}{\mathcal{D}, R, G \vdash \{p\}\langle C \rangle\{q\}}\ (\textsc{atom-r})$$

$$\frac{p \Rightarrow (E = E') \qquad \mathsf{Sta}(p, R)}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbf{return}\ E')\}\mathbf{return}\ E\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbf{skip})\}}\ (\textsc{ret})$$

Figure 7.4: Inference rules (I).

call the methods at both the concrete and abstract levels, or unsafe to do so at both levels.

Starting from the initial concrete and abstract object states related by P and $\mathcal{P}$, and with the equivalent method arguments x and y at the concrete and the abstract levels, the method body C must fulfil the abstract atomic operation C'. $\mathsf{arem}(C')$ says that the high-level code which remains to be refined at this point is C'. At the end we need to re-establish the object invariant P, and show that there is no more high-level code that needs to be refined (i.e., $\mathsf{arem}(\mathbf{skip})$), which means the method body indeed refines the specification C'. We can temporarily ignore the assertion $\blacklozenge(E_k, \ldots, E_1)$ for deadlock-freedom verification.

The last three premises of the OBJ rule checks the well-formedness of the specifications. The first one says the guarantee of one thread must implies the rely of all others, a standard requirement in rely/guarantee

$$\frac{\mathcal{D}, R, G \vdash \{p\}C_1\{r\} \qquad \mathcal{D}, R, G \vdash \{r\}C_2\{q\}}{\mathcal{D}, R, G \vdash \{p\}C_1; C_2\{q\}} \text{ (SEQ)}$$

$$\frac{\mathcal{D}, R, G \vdash \{p \wedge B\}C_1\{q\} \qquad \mathcal{D}, R, G \vdash \{p \wedge \neg B\}C_2\{q\}}{\mathcal{D}, R, G \vdash \{p\}\textbf{if } (B) \ C_1 \textbf{ else } C_2\{q\}} \text{ (IF)}$$

$$\frac{\mathcal{D}, R, G \vdash \{p\}C\{q\}}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_\Diamond\}C\{\lfloor q \rfloor_\Diamond\}} \text{ (HIDE-}\Diamond\text{)}$$

$$\frac{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_1)\}C\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_2)\}}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_1; C_3)\}C\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_2; C_3)\}} \text{ (AREM)}$$

$$\frac{\begin{array}{c}\mathcal{D}, R, G \vdash \{p\}C\{q\} \qquad R' \Rightarrow R \qquad G \Rightarrow G' \\ p' \Rightarrow p \qquad q \Rightarrow q' \qquad \mathsf{Sta}(\{p', q'\}, R) \qquad \mathsf{wffAct}(R, \mathcal{D})\end{array}}{\mathcal{D}, R', G' \vdash \{p'\}C\{q'\}} \text{ (CSQ)}$$

$$\frac{\mathcal{D}, R, G \vdash \{p\}C\{q\} \qquad x \notin fv(\mathcal{D}, R, G)}{\mathcal{D}, R, G \vdash \{\exists x.\ p\}C\{\exists x.\ q\}} \text{ (EX)}$$

$$\frac{\mathcal{D}, R, G \vdash \{p_1\}C\{q_1\} \qquad \mathcal{D}, R, G \vdash \{p_2\}C\{q_2\}}{\mathcal{D}, R, G \vdash \{p_1 \wedge p_2\}C\{q_1 \wedge q_2\}} \text{ (CONJ)}$$

$$\frac{\mathcal{D}, R, G \vdash \{p_1\}C\{q_1\} \qquad \mathcal{D}, R, G \vdash \{p_2\}C\{q_2\}}{\mathcal{D}, R, G \vdash \{p_1 \vee p_2\}C\{q_1 \vee q_2\}} \text{ (DISJ)}$$

Figure 7.5: Inference rules (II).

reasoning. In Figure 7.6 we give a simplified definition of wffAct used in the second premise. Its complete definition is given in Figure 7.9, which will be explained later when we introduce stratified actions and ⧫-tokens. $\mathsf{wffAct}(R, \mathcal{D})$ says once a definite action $\mathcal{D}_\mathsf{t}$ of a thread t is enabled it cannot be disabled by an environment step in R_t. Also such an environment step either fulfils a definite action $\mathcal{D}_{\mathsf{t}'}$ of some thread t' different from t, or preserves $\mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$ too. Together with the previous premise $G_{\mathsf{t}'} \Rightarrow R_\mathsf{t}$, this condition implies $\forall \mathsf{t}'.\, G_{\mathsf{t}'} \Rightarrow [\mathcal{D}_{\mathsf{t}'}] \vee \mathcal{D}_{\mathsf{t}'}$.

$$\mathsf{wffAct}(R, \mathcal{D}) \;\text{ iff }\; \forall \mathtt{t}.\, R_{\mathtt{t}} \Rightarrow [\mathcal{D}_{\mathtt{t}}] \wedge (\forall \mathtt{t}' \neq \mathtt{t}.\, [\mathcal{D}_{\mathtt{t}'}] \vee \mathcal{D}_{\mathtt{t}'})$$

$$\mathsf{Sta}(p, R) \;\text{ iff }\; \forall \mathfrak{S}, \mathfrak{S}', u, w, C.\; ((\mathfrak{S}, (u, w), C) \models p) \wedge ((\mathfrak{S}, \mathfrak{S}') \models R) \implies (\mathfrak{S}', (u, w), C) \models p$$

$$p \Rrightarrow q \;\text{ iff }\; \forall \mathtt{t}, \sigma, \Sigma, u, w, C, \Sigma_F.\; (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \perp \Sigma_F) \implies \exists C', \Sigma'.\; ((C, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{*} (C', \Sigma' \uplus \Sigma_F)) \wedge ((\sigma, \Sigma'), (u, w), C') \models q$$

Figure 7.6: Auxiliary defs. used in logic rules (simplified version).

Therefore, once $\mathcal{D}_{\mathtt{t}}$ is enabled, the only way to disable it is to let the thread $\mathtt{t}$ finish the action. As an example, consider the following $\mathcal{D}_{\mathtt{t}}$:

$$\mathcal{D}_{\mathtt{t}} \stackrel{\text{def}}{=} (\mathtt{L} \mapsto \mathtt{t}) \rightsquigarrow (\mathtt{L} \mapsto 0)$$

It says that whenever a thread $\mathtt{t}$ acquires the lock $\mathtt{L}$, it will finally release the lock. Then, $\mathsf{wffAct}(R, \mathcal{D})$ require that when $\mathtt{t}$ acquires $\mathtt{L}$, every step in the system either keeps $\mathtt{L}$ unchanged or releases $\mathtt{L}$. In particular, $R_{\mathtt{t}}$ keeps $\mathtt{L}$ unchanged, that is, the environment cannot update the lock when $\mathtt{L} \mapsto \mathtt{t}$.

The last premise ($P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})$) says there cannot be enabled but unfinished definite actions when the method terminates and the object invariant P is true.

The judgment $\mathcal{D}, R, G \vdash \{p \wedge \mathsf{arem}(C')\} C \{q \wedge \mathsf{arem}(\mathbf{skip})\}$ establishes a *simulation relation* between C and C', which ensures the preservation of termination when the environment guarantees the definite action $\mathcal{D}$. It also ensures the execution of C guarantees $\mathcal{D}$ too. We explain the key rules for the judgment below.

The WHL *Rule for Loops in Total Methods*

The WHL rule, shown in Figure 7.4, is the most important rule of the logic. In this subsection, we focus on the special case when the abstract specification is total, i.e., B' in the rule is true. We will defer the explanation of the general case to Subsection 7.2.4.

When B' is true, the WHL rule establishes *both* of the following properties of the loop:

(1) it cannot loop forever with $\mathcal{D}$ continuously enabled;

(2) it cannot loop forever unless the current thread is waiting for some definite actions of its environment.

The former guarantees a definite action of the current thread is *definite* to happen once it is enabled. The latter is crucial to establish the starvation-freedom.

Why definite actions are definite. The WHL verifies the loop body with a precondition p', which can be derived from the loop invariant p if B holds. Moreover, we require each iteration to consume a $\Diamond$-token if $\mathsf{Enabled}(\mathcal{D})$ holds at the beginning, as shown by the second premise (ignore the assertion Q for now). Since each thread has only a finite number of $\Diamond$-tokens, the loop must terminate if $\mathsf{Enabled}(\mathcal{D})$ is continuously true.

However, the last premise of the OBJ rule says $\mathsf{Enabled}(\mathcal{D})$ cannot be true if the method terminates. Therefore, $\mathsf{Enabled}(\mathcal{D})$ cannot be continuously true. Also recall the other two side conditions ($\mathsf{wffAct}(R, \mathcal{D})$ and $G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$) of the OBJ rule guarantee that, once $\mathsf{Enabled}(\mathcal{D})$ holds, the only way to make it false is to let the current thread finish the action.

Putting all these together, we know $\mathcal{D}$ will be finished once it is enabled, even with the interference R.

Starvation-freedom. To establish starvation-freedom, we need to find a condition Q saying the current thread is *not* blocked by others. Then the second premise requires each iteration to consume a $\Diamond$-token if Q holds at the beginning. Since the number of tokens is finite, the loop must terminate if Q always holds.

If Q is false, the current thread is blocked by others. Then the premise $(R, G \colon \mathcal{D}' \xrightarrow{f} (Q, \mathsf{true}))$ requires the thread must be waiting for its environment to perform a finite number of definite actions. Definition 7.1 shows the specialized case $(R, G \colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$. We defer the definition of the general case $(R, G \colon \mathcal{D} \xrightarrow{f} (Q, B_h))$ to Subsection 7.2.4.

Definition 7.1 (Definite Progress for Total Methods)**.**
$\mathfrak{S} \models (R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ iff the following hold for all t:

(1) either $\mathfrak{S} \models Q_{\mathsf{t}}$,
or there exists t' such that $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$;

(2) for any $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \wedge \langle \mathcal{D}_{\mathsf{t}'} \rangle$, then
$f_{\mathsf{t}}(\mathfrak{S}') < f_{\mathsf{t}}(\mathfrak{S})$;

(3) for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \vee G_{\mathsf{t}}$, then $f_{\mathsf{t}}(\mathfrak{S}') \leq f_{\mathsf{t}}(\mathfrak{S})$.

Here f is a function that maps the relational states $\mathfrak{S}$ to some metrics over which there is a well-founded order $<$.

Ignoring the index 0 above, the definition says either Q holds over $\mathfrak{S}$ or the definite action $\mathcal{D}_{\mathsf{t}'}$ of some environment thread t' is enabled. Also we require the metric $f(\mathfrak{S})$ to decrease when a definite action is performed. Besides, the metric should never increase at any step of the execution.

To ensure that the metric f decreases regardless of the time when the environment's definite actions take place, the definite progress should always hold. This is enforced by finding a stronger assertion J such that $p \wedge B \Rightarrow J$ and $\mathsf{Sta}(J, R \vee G)$ hold, that is, J is an invariant that holds at every program point of the loop. If $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ happens to satisfy the two premises, we can use $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ directly as J, but in practice it could be easier to prove the stability requirement by finding a stronger J. The definition of stability $\mathsf{Sta}(p, R)$ is given in Figure 7.6.

Notice in $(R, G : \mathcal{D}' \xrightarrow{f} (Q, \mathsf{true}))$ we can use $\mathcal{D}'$ instead of $\mathcal{D}$ to simplify the proof, as long as $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{wffAct}(R, \mathcal{D}')$ are satisfied. The premise $\mathcal{D}' \leqslant \mathcal{D}$, defined in Figure 7.3, says $\mathcal{D}'$ specifies a subset of definite actions in $\mathcal{D}$. For instance, if $\mathcal{D}$ consists of multiple definite actions and is in the form of $\mathcal{D}_1 \wedge \cdots \wedge \mathcal{D}_n$, $\mathcal{D}'$ may contain only a subset of these $\mathcal{D}_k$ $(1 \leq k \leq n)$. The way to exclude in $\mathcal{D}'$ irrelevant definite actions can simplify the proof of the condition (2) of definite progress (see Definition 7.1).

Given the definite progress condition, we know Q will be eventually true because each definite action is definite to happen. Then the loop

starts to consume ◊-tokens and needs to finally terminate, following our argument at the beginning.

More Inference Rules

Other rules in Figures 7.4 and 7.5 are mostly standard. Below we discuss the rules HIDE-◊, ATOM and ATOM-R. The rules in Figure 7.14 are for the **await** commands, which will be explained in Subsection 7.2.4.

The HIDE-◊ rule allows us to discard the tokens (by using $\lfloor \_ \rfloor_{\Diamond}$) when the termination of code C is already established, which is useful for modular verification of nested loops.

Atom rules for refinement reasoning. The ATOM rule allows us to logically execute the abstract *atomic* code simultaneously with every concrete step (let's first ignore the index k in the premises of the rule). We use $\vdash [p]C[q]$ to represent the total correctness of C in sequential separation logic. The corresponding rules are standard and elided here. We use $p \Rrightarrow q$ for the zero or multiple-step executions from the abstract code specified by p to the code specified by q, which is defined in Figure 7.6. Then, the ATOM rule allows us to execute zero-or-more steps of the abstract code with the execution of C, as long as the overall transition (including the abstract steps and the concrete steps) satisfies the relational guarantee G. We can lift C's total correctness to the concurrent setting as long as the environment consists of identity transitions only. To allow a weaker R, we can apply the ATOM-R rule later, which requires that the pre- and post-conditions be stable with respect to R.

Example: Ticket Locks

We prove the starvation-freedom of the ticket lock implementation in Figure 7.7 using our logic rules. We have informally discussed in Subsection 7.1 the verification of the counter using a ticket lock (`inc_tkL` in Figure 2.1(d)). To simplify the presentation, here we erase the increment in the critical section and focus on the progress property of the code in Figure 7.7. With an empty critical section, the code functions just as

```
1  local i, o;
2  <i := getAndInc(&next); ticket[i] := cid >;
3  o := owner;  while (i != o) { o := owner; }
4  owner := i + 1;
```

$$\mathsf{lock}(tl, n_1, n_2) \stackrel{\text{def}}{=} ((\mathtt{owner} = n_1) * (\mathtt{next} = n_2) \wedge (n_1 \leq n_2)) * \mathsf{tickets}(tl, n_1, n_2)$$

$$\mathsf{lockIrr}_\mathsf{t}(tl, n_1, n_2) \stackrel{\text{def}}{=} \mathsf{lock}(tl, n_1, n_2) \wedge (\mathsf{t} \notin tl)$$

$$P_\mathsf{t} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lockIrr}_\mathsf{t}(tl, n_1, n_2) \qquad G_\mathsf{t} \stackrel{\text{def}}{=} \mathit{Lock}_\mathsf{t} \vee \mathit{Unlock}_\mathsf{t} \qquad R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$\mathit{Lock}_\mathsf{t} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lockIrr}_\mathsf{t}(tl, n_1, n_2) \ltimes \mathsf{lock}(tl{+}{+}[\mathsf{t}], n_1, n_2 + 1)$$

$$\mathit{Unlock}_\mathsf{t} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2) \ltimes \mathsf{lockIrr}_\mathsf{t}(tl, n_1 + 1, n_2)$$

$$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \forall tl, n_1, n_2.\ \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2) \rightsquigarrow \mathsf{lockIrr}_\mathsf{t}(tl, n_1 + 1, n_2)$$

$$J_\mathsf{t} \stackrel{\text{def}}{=} \exists n_1, n_2, tl_1, tl_2.\ \mathsf{tlocked}_{tl_1, \mathsf{t}, tl_2}(n_1, \mathtt{i}, n_2) \wedge (\mathtt{o} \leq n_1)$$

$$Q_\mathsf{t} \stackrel{\text{def}}{=} \exists n_2, tl_2.\ \mathsf{lock}(\mathsf{t} :: tl_2, \mathtt{i}, n_2) \wedge (\mathtt{o} \leq \mathtt{i}) \qquad G' \stackrel{\text{def}}{=} \mathsf{Id}$$

$$f(\mathfrak{S}) = k \ \text{ iff } \ \mathfrak{S} \models (\mathtt{i} - \mathtt{owner} = k)$$

Figure 7.7: Proofs for the ticket lock (with auxiliary code in gray).

skip, so Figure 7.7 proves it is linearizable with respect to **skip**. The proofs of `inc_tkL` (including its starvation-freedom and linearizability with respect to the atomic `INC` in Figure 2.1(e)) are given in Liang and Feng (2018b).

To help specify the queue of the threads requesting the lock, we introduce an auxiliary array `ticket`. As shown in Figure 7.7, each array cell `ticket`$[i]$ records the ID of the unique thread which gets the ticket number i. Here we use `cid` for the current thread ID.

We then define $\mathsf{lock}(tl, n_1, n_2)$. It says that n_1 and n_2 are the values of `owner` and `next` respectively, and tl is the list of the threads recorded in `ticket`$[n_1]$, `ticket`$[n_1 + 1]$,...,`ticket`$[n_2 - 1]$ (as specified by $\mathsf{tickets}(tl, n_1, n_2)$). We write $\mathsf{lockIrr}_\mathsf{t}(tl, n_1, n_2)$ short for $\mathsf{lock}(tl, n_1, n_2) \wedge (\mathsf{t} \notin tl)$. That is, the thread `t` is "irrelevant" to the lock: it is not requesting the lock. The object invariant P (see the OBJ rule in Figure 7.4) is defined as lockIrr.

Figure 7.7 also defines the guarantee condition G of the code. G_t specifies the possible atomic actions of a thread `t`. Lock_t appends the thread

$\mathtt{t}$ at the end of tl of the threads requesting the lock and increments `next`. It corresponds to the code at line 2 of Figure 7.7. $\mathit{Unlock}_{\mathtt{t}}$ releases the lock by incrementing `owner` and dequeuing the thread $\mathtt{t}$ which currently holds the lock. It corresponds to the code at line 4 of Figure 7.7. The rely condition $R_{\mathtt{t}}$ includes all the $G_{\mathtt{t}'}$ made by the environment threads $\mathtt{t}'$.

Next we define the definite action $\mathcal{D}$. As we explained in Subsection 7.1, $\mathcal{D}_{\mathtt{t}}$ requires that whenever the thread $\mathtt{t}$ holds a lock with $\mathtt{owner} = n_1$, it should eventually release the lock by incrementing `owner` to $n_1 + 1$. We can prove the side conditions about well-formedness of specifications in the OBJ rule hold.

The key to verifying the loop at line 3 is to find a metric function f and prove definite progress $J \Rightarrow (R, G' \colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ for a stable J. As shown in Figure 7.7, we define $J_{\mathtt{t}}$ to say that the thread $\mathtt{t}$ is requesting the lock. The predicate $\mathsf{tlocked}_{tl_1,\mathtt{t},tl_2}(n_1, \mathtt{i}, n_2)$ has similar meanings as $\mathsf{lock}(tl_1 \mathtt{++}[\mathtt{t}] \mathtt{++} tl_2, n_1, n_2)$, but also says that the thread $\mathtt{t}$ takes the ticket number $\mathtt{i}$. $Q_{\mathtt{t}}$ specifies the case when tl_1 is empty (thus $\mathtt{owner} = \mathtt{i}$). We also strengthen the guarantee condition G' of the loop to Id, the identity transitions.

The metric function f maps each state $\mathfrak{S}$ to the difference between $\mathtt{i}$ and `owner` at that state, which describes the number of threads ahead of $\mathtt{t}$ in the waiting queue. We use the usual $<$ order on natural numbers as the associated well-founded order. Then, we can verify $J \Rightarrow (R, G' \colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$, due to the following reasons:

(1) Either Q holds, or some environment thread $\mathtt{t}'$ acquires the lock.

(2) The environment thread $\mathtt{t}'$ releases the lock in its hand by incrementing `owner`. Thus the metric $f(\mathfrak{S})$ decreases after a definite action made by the environment.

(3) Each action in R or G' either increases `owner` or keeps `owner` unchanged, so $f(\mathfrak{S})$ never increases during the loop execution.

Finally, we prove that the loop terminates with one $\Diamond$-token when Q holds or $\mathcal{D}$ is enabled. Then we can conclude linearizability and starvation-freedom of the ticket lock implementation in Figure 7.7.

7.2.3 Adding Delay for Deadlock-Free Objects

As we explained in Subsection 7.1, deadlock-free objects allow the progress of a thread to be delayed by its environment, as long as the whole system makes progress. Correspondingly, to verify deadlock-free objects, we extend our logic with a delay mechanism. First we find out the delaying actions and stratify them for objects with rollbacks where a delaying action may trigger more steps of other delaying actions. Then, we introduce ⧫-tokens (we use ⧫ here to distinguish them from ◊-tokens for loops) to bound the number of delaying actions in each method, so we avoid infinite delays without whole-system progress.

Multi-level rely/guarantee assertions. As shown in Figure 7.1, we index the rely/guarantee assertion $P \ltimes_k Q$ with a natural number k and call it a level-k action. We require $0 \leq k < \mathsf{maxL}$, where maxL is a predefined upper bound of all levels. Intuitively, $P \ltimes_k Q$ could make other threads do more actions at a level $k' < k$. Thus $P \ltimes_0 Q$ cannot make other threads do any more actions, i.e., it cannot delay other threads. $P \ltimes_1 Q$ could make other threads do more actions at level 0 but no more at level 1, thus we avoid the problem of delay-caused circular dependency discussed in Subsection 2.4.3.

To interpret the level numbers in the assertion semantics, we define $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R)$ in Figure 7.8 which assigns a level to the transition $(\mathfrak{S}, \mathfrak{S}')$, given the specification R. That is, if $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = k$, we say R views $(\mathfrak{S}, \mathfrak{S}')$ as a level-k transition. We let $k = \mathsf{maxL}$ if the transition does not satisfy R. Given the level function, we can now define the semantics of $\lfloor R \rfloor_0$, which picks out the transitions that R views as level-0 ones. For the following example R,

$$R \stackrel{\text{def}}{=} (P \ltimes_0 Q) \vee (P' \ltimes_1 Q')$$

$\lfloor R \rfloor_0$ is equivalent to $P \ltimes_0 Q$. Besides, $R \Rightarrow \lfloor R \rfloor_0$ means that R views all state transitions as level-0 ones, thus any state transitions of R should not delay the progress of other threads.

We use $(\mathfrak{S}, \mathfrak{S}', k) \models R$ as a shorthand for $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = k$ ($k < \mathsf{maxL}$). Then the implication $R \Rightarrow R'$ is redefined under this new interpretation, as shown in Figure 7.8.

$$
\begin{array}{lcl}
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),P \ltimes_k Q) & \stackrel{\text{def}}{=} & \begin{cases} k & \text{if } (\mathfrak{S},\mathfrak{S}') \models P \ltimes_k Q \\ \mathsf{maxL} & \text{otherwise} \end{cases} \\
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),[P]) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } (\mathfrak{S},\mathfrak{S}') \models [P] \\ \mathsf{maxL} & \text{otherwise} \end{cases} \\
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),\mathcal{D}) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } (\mathfrak{S},\mathfrak{S}') \models \mathcal{D} \\ \mathsf{maxL} & \text{otherwise} \end{cases} \\
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R \wedge R') & \stackrel{\text{def}}{=} & \mathsf{max}(\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R),\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R')) \\
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R \vee R') & \stackrel{\text{def}}{=} & \mathsf{min}(\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R),\mathcal{L}((\mathfrak{S},\mathfrak{S}'),R')) \\
\mathcal{L}((\mathfrak{S},\mathfrak{S}'),\lfloor R \rfloor_0) & \stackrel{\text{def}}{=} & \begin{cases} 0 & \text{if } \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = 0 \\ \mathsf{maxL} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
(\mathfrak{S},\mathfrak{S}') \models \lfloor R \rfloor_0 & \text{iff } \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = 0 \\
(\mathfrak{S},\mathfrak{S}',k) \models R & \text{iff } \mathcal{L}((\mathfrak{S},\mathfrak{S}'),R) = k \text{ and } k < \mathsf{maxL} \\
R \Rightarrow R' & \text{iff } \forall \mathfrak{S},\mathfrak{S}',k.\, ((\mathfrak{S},\mathfrak{S}',k) \models R) \Longrightarrow (\mathfrak{S},\mathfrak{S}',k) \models R'
\end{array}
$$

$$
\begin{array}{l}
u \ ::= \ (n_k,\ldots,n_1) \quad (1 \le k < \mathsf{maxL}) \\
(n'_m,\ldots,n'_1) <_k (n_m,\ldots,n_1) \ \text{ iff } \ (\forall i > k.\ (n'_i = n_i)) \wedge (n'_k < n_k) \\
(n'_m,\ldots,n'_1) \approx_k (n_m,\ldots,n_1) \ \text{ iff } \ (\forall i \ge k.\ (n'_i = n_i)) \\
u < u' \ \text{ iff } \ \exists k.\ u <_k u' \qquad\qquad u \le u' \ \text{ iff } \ u < u' \vee u = u' \\
(u,w) <_k (u',w') \ \text{ iff } \ (u <_k u') \vee (k = 0 \wedge u = u' \wedge w = w') \\
(u,w) \approx_k (u',w') \ \text{ iff } \ u \approx_k u' \wedge (k = 0 \Longrightarrow w = w')
\end{array}
$$

Figure 7.8: Levels of state transitions and tokens.

⧫-tokens in assertions. To ensure the progress of the whole system, we require the steps of delaying actions to pay ⧫-tokens. Since we allow multi-levels of transitions to delay other threads, the ⧫-tokens are stratified accordingly. Thus we introduce the new assertion $⧫(E_k,\ldots,E_1)$ in Figure 7.1, whose semantics is defined in Figure 7.2. It says the number of each level-j ⧫-tokens is *no less than* E_j. Here u is a sequence $(n_k,\ldots,n_1)$ recording the numbers of ⧫-tokens at different levels, as defined in Figure 7.8. We overload $<$ as the dictionary order for the sequence of natural numbers. The ordering over u and other related definitions are also given in Figure 7.8.

Inference Rules Revisited

To use the logic to verify deadlock-free objects, we need to first find in each object method the actions that may delay the progress of others. Since some of these actions may be further delayed by others, we assign levels to them to ensure each action can only be delayed by ones at higher levels. We specify the actions and their levels in the rely/guarantee conditions. We also need to decide an upper bound of these execution steps at each level and specify them as the number of ⧫-tokens in the precondition of each method.

Below we revisit the inference rules in Figure 7.4 and explain their use of multi-level actions and ⧫-tokens. In the OBJ rule, we specify in the precondition the number of ⧫-tokens needed for each object method. The side condition $\mathsf{wffAct}(R, \mathcal{D})$ is also redefined in Figure 7.9, which is explained below.

Decreasing ⧫-tokens at the atom rule. The thread loses ⧫-tokens when it performs an action that may delay other threads. This is required by the ATOM rule. Depending on whether the atomic command may delay others or not, we assign a level k in the premise $q' \Rrightarrow_k q$, which is redefined in Figure 7.9. Similar to $p \Rrightarrow q$ in Figure 7.6, it allows us to execute the abstract code. Now it also requires the number of ⧫-tokens at level k to be decreased if $k \geq 1$.

Note k cannot be arbitrarily chosen. The assignment of the level k to the atomic operation must be consistent with the level specification in G, as required by the third premise.

Being delayed: Increasing tokens at stability checking. When the progress of the thread t is delayed by a level-k ($k \geq 1$) action from its environment thread t', thread t could gain more ◊-tokens to do more loop iterations. It could also gain more level-k' ($k' < k$) ⧫-tokens to execute more level-k' actions. Here increasing tokens would not affect the soundness of our logic because the environment thread t' must pay a higher-level token for its higher-level delaying action. As we explained in Subsection 7.1.5, the ⧫-tokens at all levels in the whole system actually form a tuple which strictly descends along the dictionary order, ensuring the whole-system progress.

We re-define the stability $\mathsf{Sta}(p, R)$ in Figure 7.9 to reflect the possible increasing of tokens for the thread $\mathtt{t}$. We could reset w and the number at level $k' < k$ in u after the environment step R if this step is associated with a level k ($k \geq 1$).

Allowing queue jumps at definite progress and wffAct. As we explained in Subsection 7.1.4, for deadlock-free objects, the environment steps could cause queue jumps to delay the progress of the thread $\mathtt{t}$. Like starvation-free objects, the thread $\mathtt{t}$ using deadlock-free objects may wait for a queue of definite actions made by its environment. A queue jump would make the thread $\mathtt{t}$ wait for a longer queue of the environment's definite actions.

As Definition 7.1 for definite progress shows, $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ allows the thread to reset its metric $f(\mathfrak{S})$ for a queue jump when the environment step is associated with level $k \geq 1$ (i.e., it is a delaying action). In this case, although the current thread may be blocked for a longer time, the whole system must progress since a ⧫-token is paid by the environment thread for the delaying action.

Also the requirement $\mathsf{wffAct}(R, \mathcal{D})$ (used at the OBJ rule and the WHL rule) should be revised to allow queue jumps. The new definition is shown in Figure 7.9. Here we allow a queue jump to disable the definite action $\mathcal{D}$ of the thread at the head of the queue, so it is not necessary to require $\mathsf{Enabled}(\mathcal{D})$ to be preserved when the environment step is associated with level $k \geq 1$.

Example: Test-and-Set Locks

In Figure 7.10, we verify the test-and-set lock implementation explained in Subsection 7.1. Like the ticket lock proofs in Subsection 7.2.2, we simplify the code and prove it is linearizable with respect to **skip**. Here we omit the assertion $\mathsf{arem}(\mathbf{skip})$ at each line in the proof, and focus on proving deadlock-freedom of the code.

As defined at the top of Figure 7.10, the action $\mathsf{Lock}_\mathtt{t}$ (corresponding to the successful **cas** at line 3) is at level 1, which may delay other threads trying to acquire the lock. The $\mathsf{Unlock}_\mathtt{t}$ action is at level 0, which

$$\mathsf{wffAct}(R,\mathcal{D}) \;\text{ iff }\; \forall \mathsf{t}.\, \lfloor R_{\mathsf{t}} \rfloor_0 \Rightarrow [\mathcal{D}_{\mathsf{t}}] \wedge (\forall \mathsf{t}' \neq \mathsf{t}.\, [\mathcal{D}_{\mathsf{t}'}] \vee \mathcal{D}_{\mathsf{t}'})$$

$$\begin{array}{l} p \Rrightarrow_k q \;\text{ iff }\; \forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F. \\ \quad (((\sigma,\Sigma),(u,w),C) \models p) \wedge (\Sigma \perp \Sigma_F) \implies \exists u', w', C', \Sigma'. \\ \quad ((C, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (C', \Sigma' \uplus \Sigma_F)) \wedge (((\sigma,\Sigma'),(u',w'),C') \models q) \\ \quad \wedge\ (u',w') <_k (u,w) \qquad (<_k \text{ defined in Figure 7.8}) \end{array}$$

$$\begin{array}{l} \mathsf{Sta}(p,R) \;\text{ iff }\; \forall \mathfrak{S}, \mathfrak{S}', u, w, C, k. \\ \quad ((\mathfrak{S},(u,w),C) \models p) \wedge ((\mathfrak{S},\mathfrak{S}',k) \models R) \implies \exists u', w'. \\ \quad ((\mathfrak{S}',(u',w'),C) \models p) \ \wedge\ ((u',w') \approx_k (u,w)) \\ \qquad\qquad\qquad\qquad \text{where } \approx_k \text{ is defined in Figure 7.8.} \end{array}$$

Figure 7.9: Key auxiliary definitions for inference rules (final version that supersedes definitions in Figure 7.6).

$$\mathsf{locked}_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathtt{L} = \mathsf{t}) \qquad \mathsf{envLocked}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists \mathsf{t}'.\, \mathsf{locked}_{\mathsf{t}'} \wedge (\mathsf{t}' \neq \mathsf{t})$$

$$\mathsf{unlocked} \stackrel{\text{def}}{=} (\mathtt{L} = 0) \qquad \mathsf{notOwn}_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{unlocked} \vee \mathsf{envLocked}_{\mathsf{t}}$$

$$G_{\mathsf{t}} \stackrel{\text{def}}{=} \mathit{Lock}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}}$$

$$\mathit{Lock}_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{unlocked} \ltimes_1 \mathsf{locked}_{\mathsf{t}} \qquad \mathit{Unlock}_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{locked}_{\mathsf{t}} \ltimes_0 \mathsf{unlocked}$$

$$\mathcal{D}_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{locked}_{\mathsf{t}} \rightsquigarrow \mathsf{unlocked} \qquad J_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{notOwn}_{\mathsf{t}} \vee \mathsf{locked}_{\mathsf{t}}$$

$$Q_{\mathsf{t}} \stackrel{\text{def}}{=} \mathsf{unlocked} \vee \mathsf{locked}_{\mathsf{t}} \qquad f_{\mathsf{t}}(\mathfrak{S}) = \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{envLocked}_{\mathsf{t}} \\ 0 & \text{if } \mathfrak{S} \models Q_{\mathsf{t}} \end{cases}$$

$\{\mathsf{notOwn}_{\mathtt{cid}} \wedge \blacklozenge\}$
1 `local b := false;`
$\{((\neg \mathsf{b}) \wedge \mathsf{notOwn}_{\mathtt{cid}} \wedge \blacklozenge \wedge \Diamond) \vee (\mathsf{b} \wedge \mathsf{locked}_{\mathtt{cid}})\}$
2 `while (!b) {`
$\{(\mathsf{unlocked} \wedge \blacklozenge) \vee (\mathsf{envLocked}_{\mathtt{cid}} \wedge \blacklozenge \wedge \Diamond)\}$
3 `b := cas(&L, 0, cid);`
$\{(\mathsf{b} \wedge \mathsf{locked}_{\mathtt{cid}}) \vee ((\neg \mathsf{b}) \wedge \mathsf{notOwn}_{\mathtt{cid}} \wedge \blacklozenge \wedge \Diamond)\}$
4 `}`
$\{\mathsf{locked}_{\mathtt{cid}}\}$
5 `L := 0;`
$\{\mathsf{notOwn}_{\mathtt{cid}}\}$

Figure 7.10: Proofs for the TAS lock.

cannot delay other threads. Also the precondition is given a $\blacklozenge$-token, which is required to pay for the $\mathsf{Lock}_{\mathsf{t}}$ action.

The definite action $\mathcal{D}$ simply says that the thread t would eventually release the lock when it acquires the lock. It is easy to check that the

side conditions about R, G and $\mathcal{D}$ in the OBJ rule, e.g., $\mathsf{wffAct}(R, \mathcal{D})$, are satisfied.

$(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ specifies the queue of definite actions which now contains at most one environment thread. That is, the metric $f(\mathfrak{S})$ is 1 if the lock is not available, and is 0 otherwise. When an environment thread $\mathtt{t}'$ cuts in line by acquiring the lock when the lock is free, the current thread $\mathtt{t}$ has to wait for $\mathcal{D}_{\mathtt{t}'}$ before $\mathtt{t}$ itself progresses. Thus in $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ the current thread $\mathtt{t}$ can reset its metric $f(\mathfrak{S})$ when its environment acquires the lock.

The detailed proof at the bottom of Figure 7.10 shows the changes of tokens. We give the current thread one $\Diamond$-token (using the HIDE-$\Diamond$ rule) to do its loop at lines 2–4. It consumes this $\Diamond$-token at the beginning of the loop body when Q holds, as shown in the left branch of the assertion p before line 3. When Q does not hold, as shown in p's right branch, the loop does not consume the $\Diamond$-token.

Next we stabilize p. For the left branch, if an environment thread $\mathtt{t}'$ acquires the lock, which is a delaying action $\mathit{Lock}_{\mathtt{t}'}$, we let the current thread *regain* a $\Diamond$-token. The resulting state just satisfies the right branch of p. Thus p is already stable.

The current thread pays its $\blacklozenge$-token when its **cas** at line 3 succeeds (i.e., it acquires the lock), as shown in the left branch of the assertion after line 3. If the **cas** fails, the thread still has $\blacklozenge$ to acquire the lock in the future and $\Diamond$ to try one more iteration.

Another Example: Nested Locks with Rollback

To demonstrate the use of action levels, we correct the rollback code of Figure 2.4 and verify it. We assume all the methods of the object either acquire `L1` before `L2` (as in the left side of Figure 2.4), or acquire only one lock.

Stratified delaying actions. As in the TAS lock example in Subsection 7.2.3, lock acquirements are delaying actions. Here we have two locks `L1` and `L2`, and a thread may roll back and re-acquire `L1` if its environment owns `L2`. To support the rollbacks, we stratify the delaying actions and $\blacklozenge$-tokens in two levels. Acquirements of `L2` are at level 2,

$$
\begin{aligned}
G_t &\stackrel{\text{def}}{=} \mathit{Lock2}_t \vee \mathit{Lock1}_t \vee \mathit{Lock0}_t \vee \mathit{Unlock2}_t \vee \mathit{Unlock1}_t \\
\mathit{Lock2}_t &\stackrel{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L2}) \ltimes_2 \mathsf{locked}_t(\mathtt{L2})) * [\mathtt{L1} = \_] \\
\mathit{Lock1}_t &\stackrel{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L1}) \ltimes_1 \mathsf{locked}_t(\mathtt{L1})) * [\mathsf{unlocked}(\mathtt{L2})] \\
\mathit{Lock0}_t &\stackrel{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L1}) \ltimes_0 \mathsf{locked}_t(\mathtt{L1})) * [\mathsf{envLocked}_t(\mathtt{L2})] \\
\mathit{Unlock2}_t &\stackrel{\text{def}}{=} (\mathsf{locked}_t(\mathtt{L2}) \ltimes_0 \mathsf{unlocked}(\mathtt{L2})) * [\mathtt{L1} = \_] \\
\mathit{Unlock1}_t &\stackrel{\text{def}}{=} (\mathsf{locked}_t(\mathtt{L1}) \ltimes_0 \mathsf{unlocked}(\mathtt{L1})) * [\mathtt{L2} = \_] \\
\mathcal{D}_t &\stackrel{\text{def}}{=} \mathcal{D}2_t \wedge \mathcal{D}1_t \\
\mathcal{D}2_t &\stackrel{\text{def}}{=} \forall s.\ \mathsf{locked}_t(\mathtt{L2}) * (\mathtt{L1} = s) \rightsquigarrow \mathsf{unlocked}(\mathtt{L2}) * (\mathtt{L1} = s) \\
\mathcal{D}1_t &\stackrel{\text{def}}{=} \mathsf{locked}_t(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2}) \rightsquigarrow \mathsf{unlocked}(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2})
\end{aligned}
$$

Figure 7.11: Multi-level actions for the corrected code in Figure 2.4.

defined as *Lock2* in Figure 7.11, which may trigger rollbacks and more acquirements of L1. Acquirements of L1 at level 1 may delay other threads requesting L1, causing them to do more non-delaying actions, but cannot reversely trigger more level-2 actions. Thus we avoid the circular delay problem.

However, acquirements of L1 in some special cases cannot be viewed as delaying actions. Suppose L2 is acquired by an environment thread $\mathtt{t}'$ before the current thread t starts the method. Then t would continuously roll back until $\mathtt{t}'$ releases L2. It may acquire L1 infinitely often. In this case, viewing all acquirements of L1 as delaying actions would require t to pay ⧫-tokens infinitely often, and consequently require an infinite number of ⧫-tokens be assigned to the method at the beginning, which is impossible. To address the problem, we define in Figure 7.11 that acquiring L1 is a level-1 action *Lock1 only if* L2 is free. When L2 is acquired by the environment, we say the current thread is "blocked", and we view its acquirement of L1 as a non-delaying action *Lock0* at level 0.

To simplify the presentation, the definitions in Figure 7.11 follow the notations in LRG (Feng, 2009), using "$* [P]$" to mean that the actions on the irrelevant part P of the states are identity transitions.

Definite actions. There are two kinds of definite actions, which release the two locks respectively. As shown in Figure 7.11, $\mathcal{D}2$ says a thread

$\{\mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1)\}$
1 `lock L1;`
$p_1 \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * (\mathsf{unlocked}(\mathtt{L2}) \wedge \blacklozenge(1,0) \\ \qquad\qquad \vee\ \mathsf{envLocked}_{\mathsf{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1))\end{array}\right\}$
2 `local r := L2;`
$\left\{\begin{array}{l}\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L2}) \\ \wedge\ ((\mathtt{r} = 0) \wedge \blacklozenge(1,0)\ \vee\ (\mathtt{r} \neq 0) \wedge \blacklozenge(1,1) \wedge \Diamond)\end{array}\right\}$
3 `while (r != 0) {`
$p_2 \stackrel{\text{def}}{=} \left\{\begin{array}{l}\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) \\ * (\mathsf{unlocked}(\mathtt{L2}) \vee \mathsf{envLocked}_{\mathsf{cid}}(\mathtt{L2}) \wedge \Diamond) \wedge \blacklozenge(1,1)\end{array}\right\}$
4 `unlock L1;`
5 `lock L1;`
$\left\{\begin{array}{l}\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * (\mathsf{unlocked}(\mathtt{L2}) \wedge \blacklozenge(1,0) \\ \qquad\qquad \vee\ \mathsf{envLocked}_{\mathsf{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1) \wedge \Diamond)\end{array}\right\}$
6 `r := L2;`
$\left\{\begin{array}{l}\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L2}) \\ \wedge\ ((\mathtt{r} = 0) \wedge \blacklozenge(1,0)\ \vee\ (\mathtt{r} = 1) \wedge \blacklozenge(1,1) \wedge \Diamond)\end{array}\right\}$
7 `}`
$\{\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,0)\}$
8 `lock L2;`
$\{\mathsf{locked}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{locked}_{\mathsf{cid}}(\mathtt{L2})\}$
9 `unlock L2;`
10 `unlock L1;`
$\{\mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathsf{cid}}(\mathtt{L2})\}$

$Q_{\mathtt{t}} \stackrel{\text{def}}{=} (\mathsf{locked}_{\mathtt{t}}(\mathtt{L1}) \vee \mathsf{unlocked}(\mathtt{L1})) * \mathsf{unlocked}(\mathtt{L2})$

Figure 7.12: Proof outline for the corrected rollback example in Figure 2.4.

holding L2 eventually releases it, regardless of the status of the lock L1. $\mathcal{D}1$ says L1 will be definitely released when L2 is free. Note that a thread holding L1 may not be able to release the lock if it cannot acquire L2.

Proof outline for the rollback. As shown in Figure 7.12, when thread t starts the method, it is given $\blacklozenge(1,1)$, where the level-2 $\blacklozenge$-token is for doing *Lock2* and the level-1 $\blacklozenge$-token is for *Lock1*. The assertions notOwn is defined similarly as in Figure 7.10.

`lock L1` at line 1 is implemented using the TAS lock, and its detailed proof is in Figure 7.13, which will be explained later. The acquirement of L1 may or may not consume a level-1 $\blacklozenge$-token, depending on whether

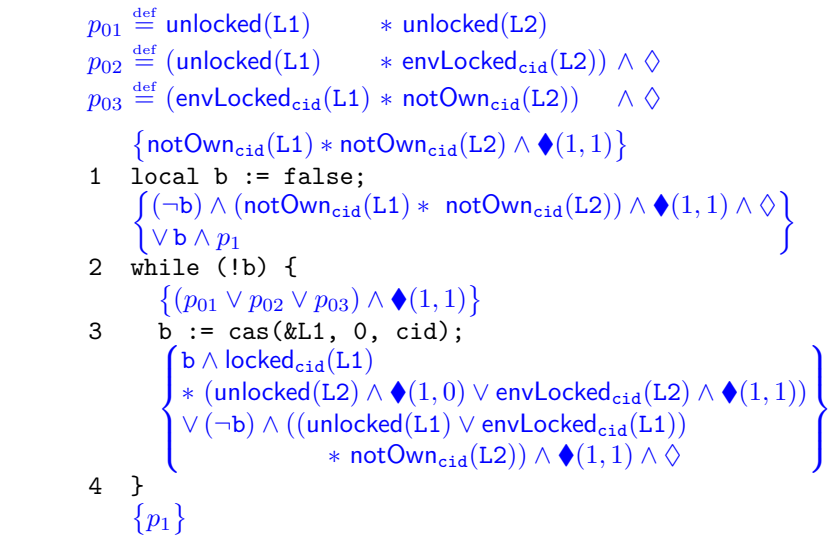

Figure 7.13: Proof outline for `lock L1` in the rollback example.

`L2` is free or not (see p_1 in Figure 7.12). If `L2` is free, line `1` is a *Lock1* action, which consumes a token. Otherwise it is a *Lock0* action and the token is not consumed, allowing the thread `t` to roll back and do *Lock1* later. Then we stabilize the assertion. For the left branch, when an environment thread acquires `L2`, i.e., the interference is at level 2, the thread `t` could re-gain the level-1 $\blacklozenge$-token, resulting in the right branch of the assertion. Stabilizing the right branch gives us the whole p_1 too. Thus p_1 is stable.

Then thread `t` tests `L2` at line `2` in Figure 7.12. When `r` is not 0, thread `t` goes into the loop at line `3`. The Q for the loop is defined at the bottom of Figure 7.12, which says that thread `t` could terminate the loop when `L1` is not acquired by the environment and `L2` is free. Before line `3`, we give the thread one $\Diamond$-token for the loop (applying the HIDE-$\Diamond$ rule). The token is consumed at the beginning of an iteration when the above Q holds. Thus, the loop body (from line `4` to line `6`) is verified with the precondition p_2. Note the thread still has one $\Diamond$-token if `L2` is not available, because the loop does not consume the token if Q does not hold. The token will be consumed at the next round when `L2` is free. On the other hand, stabilizing the left branch of the above

assertion p_2 just gives us the whole p_2: When an environment thread acquires L2, thread t could re-gain a $\Diamond$-token.

Besides, the definite progress $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ is verified as follows. When thread t is blocked (i.e., Q does not hold), there is a queue of definite actions of the environment threads. The length of the queue is at most 2, as shown by f defined below:

$$f_{\mathtt{t}}(\mathfrak{S}) \stackrel{\text{def}}{=} \begin{cases} 2 & \text{if } \mathfrak{S} \models \mathsf{envLocked}_{\mathtt{t}}(\mathtt{L2}) * \mathsf{true} \\ 1 & \text{if } \mathfrak{S} \models \mathsf{envLocked}_{\mathtt{t}}(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2}) \\ 0 & \text{if } \mathfrak{S} \models Q \end{cases}$$

When the environment thread $\mathtt{t}'$ holding L2 releases the lock (i.e., it does $\mathcal{D}2_{\mathtt{t}'}$), the queue becomes shorter. Thread t only needs to wait for the environment thread to release L1.

Acquirements of L1 in the rollback example. Finally we discuss the proof of the implementation of `lock L1` in Figure 7.13. Here we use the same Q in Figure 7.12 to verify the loop. That is, we think the thread is "blocked" if L2 is not available. Initially we give one $\Diamond$-token for the loop. Depending on whether Q holds or not, there are three cases (p_{01}, p_{02} and p_{03}) when we enter the loop. For the case p_{01}, the loop consumes the $\Diamond$-token because Q holds. For the other two cases (p_{02} and p_{03}), Q does not hold and the token is kept. Note that stabilizing p_{01} results in p_{03} when the environment acquires L1: Since L2 is free, the environment action is a level-1 delaying action *Lock1*, allowing the thread to re-gain the $\Diamond$-token.

For line 3, if b is true, we know p_{01} or p_{02} holds before the line. Depending on whether L2 is available or not, the action may or may not consume a level-1 $\blacklozenge$-token, following the same argument as in line 1 in Figure 7.12. If b is false, then p_{03} holds before the line. Stabilizing this case results in the right branch of the postcondition of line 3, with a $\Diamond$-token for the next round of loop.

It may seem strange that for the loop we do not use a $Q' \stackrel{\text{def}}{=} \mathsf{locked}_{\mathtt{t}}(\mathtt{L1}) \vee \mathsf{unlocked}(\mathtt{L1})$, i.e., the same Q as in Figure 7.10. If we use Q' instead, then the case p_{02} before line 3 cannot have the $\Diamond$-token because Q' is true in this case and the $\Diamond$-token needs to be consumed by the loop. Thus p_{02} needs to be changed to $p'_{02} \stackrel{\text{def}}{=} \mathsf{envLocked}_{\mathtt{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathtt{cid}}(\mathtt{L2})$.

Stabilizing p'_{02} can no longer give us p_{03} when the environment acquires `L1`, because the acquirement action is *Lock0* instead of *Lock1* (since `L2` is not free in this case). The thread cannot re-gain the $\Diamond$-token in p_{03}, so p_{03} cannot have the $\Diamond$-token either. As a result, we no longer have a $\Diamond$-token to pay for the next round of loop if the **cas** in line `3` fails.

7.2.4 Supporting Partial Methods

To summarize, the key ideas of LiLi to verify progress are the following:

- A thread can be blocked, relying on the actions of other threads (i.e., its environment) to make progress. To ensure it eventually progresses, we must guarantee that the environment actions that the thread waits for eventually occur.

- To avoid circularity in rely-guarantee reasoning, each thread specifies a set of *definite actions* $\mathcal{D}$, which are state transitions specified in the form of $P \rightsquigarrow Q$. The thread guarantees that, whenever a definite action $P \rightsquigarrow Q$ is "enabled" (i.e., the assertion P holds), the transition must occur so that Q eventually holds, regardless of the environment behaviors.

- A blocked thread must wait for a set of definite actions of other threads, and the size of the set must be decreasing (so that the thread is eventually unblocked).

- A thread may delay the progress of others, i.e., to make other threads to execute more steps than they need when executed in isolation. To ensure deadlock-freedom, LiLi disallows a thread to be delayed infinitely often without whole system progress. This is achieved by using tokens as resources and each delaying action must consume a token.

These ideas to reason about blocking and delay are general enough for verifying objects with partial methods, but we have to first generalize the previous rules for total methods in the following two aspects:

- Previously there was no rule for **await** commands. There **while**-loops are the only commands that affect progress. Now we have

to reason about **await** in object code, which may affect progress as well. It is interesting to see that **await** can be reasoned about similarly as **while**-loops.

- We also have $\mathbf{await}(B)\{C\}$ as partial specifications. Since we want *termination-preserving* refinement, we do not have to guarantee progress of the concrete object methods when the partial specification is disabled.

We show the new rules for **await** commands in Figure 7.14. Before discussing them, we first explain the general version of the WHL rule in Figure 7.4.

The WHL *Rule for Loops in Partial Methods*

As discussed in Subsection 7.2.2, we verify the loop body with a precondition p', which needs to be derived from the loop invariant p and the loop condition B. In two cases we must ensure that there are no infinite loops:

- the definite action $\mathcal{D}$ is enabled (see Figure 7.1 for the definition of $\mathsf{Enabled}(\mathcal{D})$). Then the loop must terminate to guarantee that the definite action $\mathcal{D}$ definitely occurs.
- the current thread is not blocked. Here we need to find a condition Q that ensures the current thread can make progress without waiting for actions of other threads.

The second premise of the rule says, in either of the two cases above we must consume a $\Diamond$-token for each round of the loop, as p' has one less token than $p \wedge B$.

On the other hand, if the current thread is blocked (Q does not hold) and it is not in the middle of a definite action, the loop can run an indefinite number of rounds to wait for the environment actions. It does not have to consume tokens. However, we must ensure the thread cannot be blocked forever, i.e., Q cannot be always false. This is achieved by the *definite progress* condition. We have explained the definite progress condition for total methods in Definition 7.1. Below

$$\frac{\begin{array}{c} p \wedge \mathsf{Enabled}(\mathcal{D}) \Rightarrow B \qquad \mathcal{D}, \mathsf{Id}, G \vdash \{p \wedge B\}\langle C\rangle\{q\} \\ \mathsf{Sta}(\{p,q\}, R) \qquad \mathcal{D}' \leqslant \mathcal{D} \qquad \mathsf{wffAct}(R, \mathcal{D}') \\ p \Rightarrow \exists B', C'.\, \mathsf{arem}(\mathbf{await}(B')\{C'\}) \wedge (R\colon \mathcal{D}' \circ\!\!\xrightarrow{f} (B, B')) \end{array}}{\mathcal{D}, R, G \vdash_{\mathsf{wfair}} \{p\}\mathbf{await}(B)\{C\}\{q\}} \ (\textsc{await-w})$$

$$\frac{\begin{array}{c} p \wedge \mathsf{Enabled}(\mathcal{D}) \Rightarrow B \qquad \mathcal{D}, \mathsf{Id}, G \vdash \{p \wedge B\}\langle C\rangle\{q\} \\ \mathsf{Sta}(\{p,q\}, R) \qquad \mathcal{D}' \leqslant \mathcal{D} \qquad \mathsf{wffAct}(R, \mathcal{D}') \\ p \Rightarrow \exists B', C'.\, \mathsf{arem}(\mathbf{await}(B')\{C'\}) \wedge (R\colon \mathcal{D}' \bullet\!\!\xrightarrow{f} (B, B')) \end{array}}{\mathcal{D}, R, G \vdash_{\mathsf{sfair}} \{p\}\mathbf{await}(B)\{C\}\{q\}} \ (\textsc{await-s})$$

Figure 7.14: Inference rules (III).

we show the generalized definition for partial methods in Definition 7.2, with the changes highlighted in gray boxes.

Definition 7.2 (Definite Progress). $\mathfrak{S} \models (R, G\colon \mathcal{D} \xrightarrow{f} (Q, B_h))$ iff the following hold for any t:

(1) either $\mathfrak{S} \models Q_{\mathsf{t}}$, or $\mathfrak{S} \models \neg B_h$, or there exists t' such that $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$;

(2) for any $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \wedge (\langle \mathcal{D}_{\mathsf{t}'} \rangle \vee ((\neg B_h) \ltimes B_h))$, then $f_{\mathsf{t}}(\mathfrak{S}') < f_{\mathsf{t}}(\mathfrak{S})$;

(3) for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \vee G_{\mathsf{t}}$, then $f_{\mathsf{t}}(\mathfrak{S}') \leq f_{\mathsf{t}}(\mathfrak{S})$.

Here f is a function that maps the relational states $\mathfrak{S}$ to some metrics over which there is a well-founded order $<$.

The definite progress condition $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, B_h))$ tries to ensure Q is eventually always true, unless B_h is eventually always false. It requires the following conditions to hold:

(1) Either Q holds, which means that the low-level code is no longer blocked; or the high-level specification $\mathbf{await}(B_h)\{C'\}$ is disabled, so that the low-level code does not have to progress to refine the high-level code; or one of the definite actions in $\mathcal{D}$ that the current thread t waits for is enabled in some thread t'. Here $\mathcal{D}$ can be

viewed as a set of n definite actions in the form of $\mathcal{D}_1 \wedge \cdots \wedge \mathcal{D}_n$ parameterized with thread IDs.

(2) There is a well-founded metric f that becomes strictly smaller whenever (a) an environment thread t' executes a definite action in $\mathcal{D}$, or (b) an environment action has turned the high-level command from disabled to enabled. Case (a) requires that the number of definite actions waited by the current thread must be strictly decreasing. Therefore eventually there are no enabled definite actions. By condition (1) we know eventually either Q or $\neg B_h$ is true. Case (b) further requires that the high-level command cannot be infinitely often disabled and then enabled during the loop. Therefore either B_h is eventually always true or it is eventually always false. In the former case we know Q must be eventually always true by the above condition (1). In the latter the loop does not have to terminate because the execution is well-blocked (see Figure 5.3).

(3) The value of f over program states cannot be increased by any level-0 actions (i.e., non-delaying actions).

Note that the last two conditions do not prevent delaying actions (level-k actions where $k > 0$) from increasing the value of f, but such an increase can only occur a finite number of times because each delaying action consumes a ⧫-token. The effects of delaying actions are shown in the ATOM rule in Figure 7.4, which has been discussed in Subsection 7.2.3.

As explained in Subsection 7.2.2, to ensure the definite progress condition always holds, we need to find an invariant J which is preserved by any program step (by the current thread or by the environment), and require that J implies definite progress, given the currently remaining high-level command $\mathbf{await}(B')\{C'\}$. Note that to simplify the presentation we treat $\mathsf{arem}(\mathbf{skip})$ as $\mathsf{arem}(\mathbf{await}(\mathsf{true})\{\mathbf{skip}\})$ so that it can be reasoned about in the same way.

Rules for **await** *Commands*

Now we introduce the two new rules, AWAIT-W and AWAIT-S, to verify **await** commands in the *object implementation* under weakly fair and

strongly fair scheduling. We use the subscripts of the judgment to distinguish the scheduling.

Naturally the AWAIT-W rule combines the ATOM rule and the WHL rule. If $\mathbf{await}(B)\{C\}$ is enabled, we can simply treat C as an atomic block $\langle C\rangle$ and apply the ATOM rule to verify it. In this case we do not need to consider the interference and take Id as the rely condition (remember that Id is a shorthand notation for $[\mathsf{true}]$, which specifies arbitrary identity transitions).

Similar to the WHL rule, if the definite action $\mathcal{D}$ is enabled, then $\mathbf{await}(B)\{C\}$ must be enabled at this point (see the first premise of the AWAIT-W rule). This is because we require that, when enabled, the definite action $\mathcal{D}$ must be fulfilled regardless of environment behaviors. Therefore the current thread cannot be blocked.

Finally, we require that, even if the command is blocked, it must be eventually enabled unless the corresponding high-level specification is blocked too. So if we view the enabling condition B the same as the condition Q we use in the WHL rule, we require the same *definite progress* condition, except that things are simpler here because $\mathbf{await}(B)\{C\}$ finishes in one step once enabled, unlike loops which take multiple steps to finish even if Q holds. Therefore we do not need the invariant J used in the WHL rule, and we do not need to consider actions in G in the definite progress condition. We can use a simpler condition $(R\colon \mathcal{D}'\circ\!\xrightarrow{f}(B,B'))$ defined below, which simply instantiates G with Id and Q with B in $(R,G\colon \mathcal{D}'\xrightarrow{f}(Q,B'))$ (see Definition 7.2).

Definition 7.3 (Definite Progress for Await)**.**

- $\mathfrak{S}\models(R\colon \mathcal{D}\circ\!\xrightarrow{f}(B_l,B_h))$ iff $\mathfrak{S}\models(R,\mathsf{Id}\colon \mathcal{D}\xrightarrow{f}(B_l,B_h))$.
- $\mathfrak{S}\models(R\colon \mathcal{D}\bullet\!\xrightarrow{f}(B_l,B_h))$ iff the following hold for any t:
  (1) either $\mathfrak{S}\models B_l$, or $\mathfrak{S}\models\neg B_h$, or there exists t' such that $\mathsf{t}'\neq\mathsf{t}$ and $\mathfrak{S}\models\mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$;
  (2) for any $\mathsf{t}'\neq\mathsf{t}$ and $\mathfrak{S}'$, if $(\mathfrak{S},\mathfrak{S}',0)\models R_{\mathsf{t}}\wedge(\langle\mathcal{D}_{\mathsf{t}'}\rangle\vee((\neg B_h)\ltimes B_h))$, then $f_{\mathsf{t}}(\mathfrak{S}')<f_{\mathsf{t}}(\mathfrak{S})$;
  (3) for any $\mathfrak{S}'$, if $(\mathfrak{S},\mathfrak{S}',0)\models R_{\mathsf{t}}\wedge((\neg B_l)\ltimes(\neg B_l))$, then $f_{\mathsf{t}}(\mathfrak{S}')\leq f_{\mathsf{t}}(\mathfrak{S})$.

The AWAIT-S rule for strongly fair scheduling looks almost the same as the AWAIT-W rule, with a slightly different definite progress condition $(R : \mathcal{D}' \bullet \xrightarrow{f} (B, B'))$, which is also shown in Definition 7.3 with the difference highlighted in the gray box. The key difference here is that the low-level enabling condition B (represented as B_l in Definition 7.3) does not have to be stable once it becomes true. Under strongly fair scheduling we know the **await** block will be executed as long as it is enabled infinitely often. Therefore in condition (3) we only need to ensure that f does not increase if the enabling condition B_l remains false, but we allow f to increase whenever we see B_l holds.

The (general) WHL rule and the AWAIT-W and AWAIT-S rules are the only new command rules we introduce to reason about partial methods and blocking primitives. All the other rules in Figures 7.4 and 7.5 can be used for partial methods without any changes.

7.3 Soundness

The two AWAIT rules actually give us two program logics, for strongly fair and weakly fair scheduling respectively. To distinguish them, we use $\mathcal{D}, R, G \vdash_\chi \{P\}\Pi{:}\Gamma$ to represent the verification using the logic for χ-scheduling ($\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$), where the corresponding AWAIT rule is used.

Theorem 7.1 shows that our logic is sound in that it guarantees linearizability and progress properties of concurrent objects.

Theorem 7.1 (Soundness). If $\mathcal{D}, R, G \vdash_\chi \{P\}\Pi : \Gamma$ and $\varphi \Rightarrow P$, then

(1) $\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma$ (i.e., linearizability holds); and

(2) $\mathsf{PDF}^{\chi}_{\varphi,\Gamma}(\Pi)$; and

(3) if $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\mathsf{PSF}^{\chi}_{\varphi,\Gamma}(\Pi)$ holds; and

(4) if $\mathsf{total}(\Gamma)$, then $\mathsf{DF}_{\varphi}(\Pi)$; and

(5) if $\mathsf{total}(\Gamma)$, $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\mathsf{SF}_{\varphi}(\Pi)$ holds; and

(6) if $\mathsf{total}(\Gamma)$ and $\mathcal{D}$ is instantiated to $\mathsf{false} \rightsquigarrow \mathsf{true}$, then $\mathsf{LF}_{\varphi}(\Pi)$ holds; and

	non-delay		*delay*
non-blocking	wait-freedom	$\Rightarrow$	lock-freedom
	$\Downarrow$		$\Downarrow$
blocking	starvation-freedom	$\Rightarrow$	deadlock-freedom

Figure 7.15: Progress properties for total methods.

(7) if $\mathsf{total}(\Gamma)$, $\mathcal{D}$ is instantiated to $\mathsf{false} \rightsquigarrow \mathsf{true}$, $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\mathsf{WF}_\varphi(\Pi)$ holds.

Here $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$, and $\varphi \Rightarrow P \stackrel{\text{def}}{=} \forall \sigma, \Sigma.\, (\varphi(\sigma) = \Sigma) \Longrightarrow (\sigma, \Sigma) \models P$.

Theorem 7.1 says that LiLi ensures linearizability and partial deadlock freedom (PDF) together. It also ensures partial starvation freedom (PSF) if the rely/guarantee conditions specify only level-0 actions, as required by $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$. That is, none of the object actions of a thread could delay the progress of other threads. With the specialized R and G, we can derive the progress of each single thread, which gives us PSF.

When the atomic specification Γ is total (i.e., each method body in Γ is an atomic block $\langle C \rangle$ followed by a **return** E command), LiLi ensures the four progress properties for total methods (see items (4)–(7) in Theorem 7.1). It ensures deadlock-freedom (DF), and when R and G specify only level-0 actions, it also ensures starvation-freedom (SF). To verify lock-freedom (LF) and wait-freedom (WF), we instantiate the definite actions $\mathcal{D}$ to $\mathsf{false} \rightsquigarrow \mathsf{true}$.

To understand the soundness theorem, Figure 7.15 shows the relationships among all the four progress properties for total methods (where "$\Rightarrow$" represents implications). We sort them in two dimensions: *blocking* and *delay* (their difference has been explained in Subsection 2.4.3). Starvation-free or deadlock-free objects allow a thread to be blocked, and lock-free and deadlock-free objects permit delay.

LiLi handles blocking by definite actions, and supports delay by ♦-tokens and multi-level actions. By ignoring either or both features, it can be instantiated to verify objects with any of the four progress properties in Figure 7.15.

For instance, to verify lock-free objects, we instantiate the definite actions $\mathcal{D}$ to be $\mathsf{false} \rightsquigarrow \mathsf{true}$, and use only the supports for delay.

Then a thread cannot rely on the environment threads' $\mathcal{D}$, meaning that it is never blocked. The WHL rule in Figure 7.4 is reduced to the following rule, requiring that the loop terminates (the $\Diamond$-tokens decrease at each iteration) unless being delayed by the environment. The definite progress condition $J \Rightarrow (R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathsf{true}))$ could trivially hold by setting both Q and J to be true and f to be a constant function.

$$\frac{p \wedge B \Rightarrow p' * (\Diamond \wedge \mathsf{emp}) \qquad \mathsf{false} \rightsquigarrow \mathsf{true}, R, G \vdash \{p'\}C\{p\}}{\mathsf{false} \rightsquigarrow \mathsf{true}, R, G \vdash \{p\}\mathbf{while}\ (B)\{C\}\{p \wedge \neg B\}}\ (\text{WHL-LFWF})$$

To verify wait-free objects, besides instantiating $\mathcal{D}$ as $\mathsf{false} \rightsquigarrow \mathsf{true}$, we also require R and G to specify actions of level 0 only, as in Theorem 7.1(3). The instantiation results in the logic rules disallowing both blocking and delay, so we know every method would terminate regardless of the environment interference.

To prove Theorem 7.1, we first prove the logic establishes the progress-aware contextual refinements, and then apply the Abstraction Theorem 6.1 to ensure linearizability and the progress properties.

7.4 Examples

We have seen a few small examples showing the use of LiLi. Below we give an overview of other algorithms we have verified.

- *PSF and PDF algorithms.*
    - *Locks and other partial objects.* We have applied LiLi to verify ticket locks (Mellor-Crummey and Scott, 1991), test-and-set locks (Herlihy and Shavit, 2008), bounded partial queues with two locks (Herlihy and Shavit, 2008) (where the locks are implemented using the specification (2.3.2)) and Treiber stacks (Treiber, 1986) with partial `pop` methods.
    - *Wrappers.* Perhaps interestingly, we also use our logic to prove that, for the atomic partial specification Γ for locks, the wrapping of Γ (as the object implementation) respects Γ itself as the atomic specification under the designated fairness conditions, i.e., $(\mathcal{D}, R, G \vdash \{P\}\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma) : \Gamma)$ holds

for certain $\mathcal{D}$, R, G and P, and for different combinations of fairness χ and progress Prog. This result validates our wrappers and program logic. It shows $\mathsf{Prog}^{\chi}_{\varphi,\Gamma}(\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma))$ holds, i.e., each wrapper itself satisfies the corresponding progress property.

- *Starvation-free and deadlock-free algorithms.*
  - *Coarse-grained synchronization.* The easiest way to implement a SF or DF concurrent object is using a single lock to protect all the object data. As an example, we have verified the counter with various lock implementations (Herlihy and Shavit, 2008; Mellor-Crummey and Scott, 1991), including ticket locks, Anderson array-based queue locks, CLH list-based queue locks, MCS list-based queue locks, and TAS locks. We show that the coarse-grained object with ticket locks or queue locks is starvation-free, and it is deadlock-free with TAS locks.
  - *Fine-grained and optimistic synchronization.* As examples with more permissive locking scheme, we have verified Michael-Scott two-lock queues (Michael and Scott, 1996), lock-coupling lists (Herlihy and Shavit, 2008), optimistic lists (Herlihy and Shavit, 2008), and lazy lists (Heller *et al.*, 2005). We show that the two-lock queues and the lock-coupling lists are starvation-free if all their locks are implemented using ticket locks, and they are deadlock-free if their locks are TAS locks. The optimistic lists and the lazy lists have rollback mechanisms, and we prove they are deadlock-free.
- *Lock-free algorithms.* We have also verified several variants of **cas**-based lock-free counters, Treiber stacks (Treiber, 1986), Michael-Scott lock-free queues (Michael and Scott, 1996) and DGLM lock-free queues (Doherty *et al.*, 2004).

To the best of our knowledge, we are the first to formally verify the starvation-freedom of lock-coupling lists, the deadlock-freedom of optimistic lists and lazy lists, and the PSF/PDF of the lock algorithms.

In the following subsections, we first verify the optimistic list algorithm. Then we show the verification of test-and-set locks and ticket locks. Finally, to demonstrate the use of the rules for **await** commands, we verify simple locks implemented using **await** which guarantee PSF under weak fairness.

7.4.1 Optimistic Lists

Below we verify the optimistic list-based implementation in Figure 2.3 of a mutable set data structure. The algorithm has operations `add`, which adds an element to the set, and `rmv`, which removes an element from the set. Figure 7.16 shows the code and the proof outline for `rmv`.

We have informally explained the idea of the algorithm in Subsection 7.1.5. To verify its progress in LiLi, we need to recognize the delaying actions, specify them in rely and guarantee conditions with appropriate levels, define the definite actions, and finally prove the termination of loops following the WHL rule.

Following the earlier linearizability proofs in RGSep (Vafeiadis, 2008), the basic actions of a thread include the lock acquire (line `4`) and release actions (lines `6` and `12`), and the *Add* and *Rmv* actions (lines `8`–`11`) that insert and delete nodes from the list respectively. Since we use TAS locks here, acquirements of a lock will delay other threads competing for the same lock. Thus lock acquirements are delaying actions, as illustrated in Subsection 7.2.3. Also, the *Add* and *Rmv* actions may cause the failure of the validation (at line `5`) in other threads. The failed validation will further cause the threads to roll back and to acquire the locks again. Therefore the *Add* and *Rmv* actions are also delaying actions that may lead to more lock acquirements. In our rely and guarantee specifications, the *Add* and *Rmv* actions are level-2 delaying actions, while lock acquirements are at level 1.

Next we define the definite actions $\mathcal{D}$ that a blocked thread may wait for. Since a thread is blocked only if the lock it tries to acquire is unavailable, we only need to specify in $\mathcal{D}$ the various scenarios under which the lock release would definitely happen. The definitions are omitted here.

```
rmv(int e) {
    {♦(1,2) ∧ arem(RMV(e)) ∧ ...}
 1  local b := false, p, c, n;
    {¬b ∧ ♦(1,2) ∧ ◇ ∧ ... ∨ b ∧ ...}
 2  while (!b) {
      {♦(1,2) ∧ ...}
 3    (p, c) := find(e); // a loop of list traversal
      {valid(p,c) ∧ ♦(1,2) ∧ ... ∨ invalid(p,c) ∧ ♦(1,4) ∧ ◇ ∧ ...}
 4    lock p; lock c;
      {valid(p,c) ∧ ♦(1,0) ∧ ... ∨ invalid(p,c) ∧ ♦(1,2) ∧ ◇ ∧ ...}
 5    b := validate(p, c); // a loop of list traversal
 6    if (!b) { unlock c; unlock p; }
      {b ∧ valid(p,c) ∧ ♦(1,0) ∧ ... ∨ ¬b ∧ ♦(1,2) ∧ ◇ ∧ ...}
 7  }
    {valid(p,c) ∧ ♦(1,0) ∧ arem(RMV(e)) ∧ ...}
 8  if (c.data = e) {
 9    n := c.next;
      {valid(p,c,e,n) ∧ ♦(1,0) ∧ arem(RMV(e)) ∧ ...}
10    < p.next := n;  gn := gn ∪ {c} >; // LP
      {valid(p,n) ∧ arem(skip) ∧ ...}
11  }
12  unlock c; unlock p;
    {arem(skip) ∧ ...}
}
```

Figure 7.16: Proofs for optimistic lists (with auxiliary code in gray).

We also need to find a metric f to prove the definite progress condition in the WHL rule. We define f as the number of all the locked nodes, including those on the list and those that have been removed from the list but have not been unlocked yet. It is a conservative upper bound of the length of the queue of definite actions that a blocked thread is waiting for. It is easy to check that every definite action $\mathcal{D}$ makes the metric to decrease, and that a thread is unblocked to acquire the lock when the metric becomes 0.

In Figure 7.16, the precondition is given $\blacklozenge(1,2)$, two level-1 $\blacklozenge$-tokens for locking two adjacent nodes, and one level-2 $\blacklozenge$-token for doing *Rmv*. We apply the (HIDE-$\Diamond$) rule and assign one $\Diamond$-token to the loop at lines 2–7, so the loop should terminate in one round if it is not delayed by the environment.

A round of loop is started at the cost of the ◊-token. The code `find` at line 3 traverses the list. After line 3, `p` and `c` may be valid: both of them are on the list and `p.next` is `c`. However, if the environment updates the list by the level-2 delaying actions *Add* or *Rmv*, the two nodes `p` and `c` may no longer satisfy valid. In this case, invalid(`p`, `c`) holds, and the current thread could gain two more level-1 ⧫-tokens and one more ◊-token, allowing it to roll back and re-lock the nodes in a new round.

At line 4, `lock p` and `lock c` consume two level-1 ⧫-tokens respectively. The validation at line 5 succeeds in a valid state, and fails in an invalid state. Thus we can re-establish the loop invariant after line 6.

Lines 8–11 perform the node removal. Line 10 is the linearization point (LP in the figure), at which we fulfill the abstract atomic operation `RMV(e)`. Afterwards, the remaining abstract code becomes **skip**. To help specify the shared state, in line 10 we introduce an auxiliary variable `gn` to collect the locations of removed nodes.

Due to space limit, here we only give a brief overview of the proofs and omit many details, including the specifications of rely and guarantee conditions, definite actions, and the proofs of the implementation of `find` (line 3), `validate` (line 5) and `lock` (line 4).

7.4.2 Test-and-Set Locks

In Subsection 7.2.3, we have verified DF of an object using test-and-set locks (see Figure 7.10). In Figure 7.17, we verify PDF of the test-and-set lock object with respect to the atomic partial specifications `L_ACQ'` and `L_REL` defined in (2.3.2). To distinguish the variables at the two levels, below we use capital letters (e.g., `L`) in the specifications and small letters (e.g., `l`) in the implementations.

As we explained in Subsection 3.1, the method `L_rel` and the specification `L_REL` have annotated preconditions (`l` = `cid`) and (`L` = `cid`), respectively. That is, it is not allowed to call `L_rel` (or `L_REL`) when the thread does not hold the lock. The annotated precondition for `L_acq` and `L_ACQ'` is true. In Figure 7.17, we define the assertion lock as the object invariant P used in the OBJ rule. Then the method `L_acq` is verified with the precondition lock, and `L_rel` is verified with the

$$\mathsf{lock}_s \stackrel{\text{def}}{=} (\mathtt{l} = \mathtt{L} = s) \qquad \mathsf{lock} \stackrel{\text{def}}{=} \exists s.\ \mathsf{lock}_s$$
$$\mathsf{locked}_\mathsf{t} \stackrel{\text{def}}{=} \mathsf{lock}_\mathsf{t} \wedge (\mathsf{t} \neq 0) \qquad \mathsf{locked} \stackrel{\text{def}}{=} \exists \mathsf{t}.\ \mathsf{locked}_\mathsf{t}$$
$$\mathsf{unlocked} \stackrel{\text{def}}{=} \mathsf{lock}_0$$
$$R_\mathsf{t} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'} \qquad G_\mathsf{t} \stackrel{\text{def}}{=} \mathit{Acq}_\mathsf{t} \vee \mathit{Rel}_\mathsf{t}$$
$$\mathit{Acq}_\mathsf{t} \stackrel{\text{def}}{=} \mathsf{unlocked} \ltimes_1 \mathsf{locked}_\mathsf{t} \qquad \mathit{Rel}_\mathsf{t} \stackrel{\text{def}}{=} \mathsf{locked}_\mathsf{t} \ltimes_0 \mathsf{unlocked}$$
$$\mathcal{D} \stackrel{\text{def}}{=} \mathsf{false} \rightsquigarrow \mathsf{true}$$
$$J \stackrel{\text{def}}{=} \mathsf{lock} \qquad Q \stackrel{\text{def}}{=} \mathsf{unlocked} \qquad f(\mathfrak{S}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{locked} \\ 0 & \text{if } \mathfrak{S} \models Q \end{cases}$$

```
L_acq(){
   {lock ∧ ♦ ∧ arem(L_ACQ')}
1  local b := false;
   {((¬b) ∧ lock ∧ ♦ ∧ ◇ ∧ arem(L_ACQ')) ∨ (b ∧ locked_cid ∧ arem(skip))}
2  while (!b) {
     {((unlocked ∧ ♦) ∨ (locked ∧ ♦ ∧ ◇)) ∧ arem(L_ACQ')}
3    b := cas(&l, 0, cid);
4  }
   {locked_cid ∧ arem(skip)}
}
L_rel(){
   {locked_cid ∧ arem(L_REL)}
5  l := 0;
   {lock ∧ arem(skip)}
}
```

Figure 7.17: Proofs for the test-and-set lock.

precondition $\mathsf{lock} \wedge (\mathtt{l} = \mathtt{cid})$ which is reduced to $\mathsf{locked}_\mathtt{cid}$, as shown in Figure 7.17.

To verify `L_acq`, we make the following key observations. When the **cas** at line 3 succeeds, `L_ACQ'` must be enabled and can be executed correspondingly. And at the time when the **cas** fails, `L_ACQ'` must be disabled. The progress of `L_acq` relies on that the environment thread holding the lock could eventually release the lock, i.e., turning the current thread's `L_ACQ'` from disabled to enabled. But such an action is not "definite", since the client thread may never call the `L_rel` method. The definite action $\mathcal{D}$ for this object can be defined as $\mathsf{false} \rightsquigarrow \mathsf{true}$, saying that there is no definite action that a thread needs to complete.

The action $\mathit{Acq}_\mathtt{t}$ (corresponding to the successful **cas** at line 3) is a delaying action (defined with level 1). When thread t succeeds in **cas**, termination of other threads' L_acq can be delayed, as allowed by PDF. The thread t has to pay a ♦-token, given in the precondition of L_acq.

The definite progress condition $(R, G : \mathcal{D} \xrightarrow{f} (Q, \mathtt{L{=}0}))$ now says that thread t is either at a state that it itself can progress (i.e., Q holds), or blocked at the abstract level (i.e., L=0 does not hold). The metric $f_\mathtt{t}(\mathfrak{S})$ decreases when an environment thread releases L, but can be reset (which means thread t is delayed) if an environment thread successfully acquires the lock.

By the Soundness Theorem 7.1, we know the test-and-set lock object satisfies the PDF property, and contextually refines the abstraction generated by the corresponding PDF wrappers in Figure 6.3, under strongly and weakly fair scheduling.

7.4.3 Ticket Locks

In Subsection 7.2.2, we have verified SF of an object using ticket locks (see Figure 7.7). In Figure 7.18, we prove the ticket lock object itself satisfies PSF. We introduce some write-only auxiliary variables to help the verification. First, as in the proofs in Figure 7.7 in Subsection 7.2.2, we introduce an array ticket to help specify the queue of the threads requesting the lock. Each array cell ticket[i] records the ID of the unique thread getting the ticket number i (see line 2). Second, we introduce a lock bit l to make the lock acquirement and lock release explicit (see lines 4 and 5).

We then define the object invariant $\mathsf{lock}(s, tl, n_1, n_2)$. It says that the lock bits l and L are equal, n_1 and n_2 are the values of owner and next respectively, and tl is the list of the threads recorded in $\mathtt{ticket}[n_1]$, $\mathtt{ticket}[n_1+1], \ldots, \mathtt{ticket}[n_2-1]$ (as specified by $\mathsf{tickets}(tl, n_1, n_2)$).

The guarantee condition $G_\mathtt{t}$ describes the possible atomic actions of thread t. $\mathit{Req}_\mathtt{t}$ adds t at the end of tl of the threads requesting the lock and also increments next. It corresponds to line 2 in the code at the top of Figure 7.18. $\mathit{Acq}_\mathtt{t}$ sets the lock bits to t, explicitly indicating the lock acquirement (see line 4). It is also a definite action (see the definition of $\mathcal{D}_\mathtt{t}$) since thread t must acquire the lock if its loop at

```
tkL_acq(){
1  local i, o;
2  <i := getAndInc(&next); ticket[i] := cid >;
3  o := owner;  while (i != o) { o := owner; }
4  l := cid;
}
tkL_rel(){
5  <owner := owner+1; l := 0 >;
}
```

$$\mathsf{lock}(s, tl, n_1, n_2) \stackrel{\text{def}}{=} (\mathtt{l} = \mathtt{L} = s \wedge (s = \mathtt{head}(tl) \vee s = 0)) * ((\mathtt{owner} = n_1) * (\mathtt{next} = n_2) \wedge (n_1 \leq n_2)) * \mathsf{tickets}(tl, n_1, n_2)$$

$$G_\mathsf{t} \stackrel{\text{def}}{=} \mathit{Req}_\mathsf{t} \vee \mathit{Acq}_\mathsf{t} \vee \mathit{Rel}_\mathsf{t}$$

$$\mathit{Req}_\mathsf{t} \stackrel{\text{def}}{=} \exists s, tl, n_1, n_2.\ \mathsf{lock}(s, tl, n_1, n_2) \ltimes \mathsf{lock}(s, tl{+}{+}[\mathsf{t}], n_1, n_2 + 1)$$

$$\mathit{Acq}_\mathsf{t} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lock}(0, \mathtt{t} :: tl, n_1, n_2) \ltimes \mathsf{lock}(\mathtt{t}, \mathtt{t} :: tl, n_1, n_2)$$

$$\mathit{Rel}_\mathsf{t} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lock}(\mathtt{t}, \mathtt{t} :: tl, n_1, n_2) \ltimes \mathsf{lock}(0, tl, n_1 + 1, n_2)$$

$$\mathcal{D}_\mathsf{t} \stackrel{\text{def}}{=} \forall tl, n_1, n_2.\ \mathsf{lock}(0, \mathtt{t} :: tl, n_1, n_2) \rightsquigarrow \mathsf{lock}(\mathtt{t}, \mathtt{t} :: tl, n_1, n_2)$$

$$J_\mathsf{t} \stackrel{\text{def}}{=} \exists s, n_1, n_2, tl_1, tl_2.\ \mathsf{tlocked}_{tl_1, \mathtt{t}, tl_2}(s, n_1, \mathtt{i}, n_2) \wedge (\mathtt{o} \leq n_1)$$

$$Q_\mathsf{t} \stackrel{\text{def}}{=} \exists n_2, tl_2.\ \mathsf{lock}(0, \mathtt{t} :: tl_2, \mathtt{i}, n_2) \wedge (\mathtt{o} \leq \mathtt{i})$$

$$f(\mathfrak{S}) \stackrel{\text{def}}{=} \begin{cases} 2k+1 & \text{if } \mathfrak{S} \models (\mathtt{i} - \mathtt{owner} = k) * (\mathtt{l} = 0) \\ 2k & \text{if } \mathfrak{S} \models (\mathtt{i} - \mathtt{owner} = k) * (\mathtt{l} \neq 0) \end{cases}$$

Figure 7.18: Proofs for the ticket lock (with auxiliary code in gray).

line 3 terminates. Rel_t increments `owner` to dequeue the thread `t` which currently holds the lock, and resets the lock bits (see line 5). All actions are at level 0. There are no delaying actions.

By applying the WHL rule of our logic, we need to prove the definite progress condition $J \Rightarrow (R, \mathsf{Id} : \mathcal{D} \xrightarrow{f} (Q, \mathtt{L{=}0}))$ for the loop at line 3. Here J, Q and f are defined at the bottom of Figure 7.18. In the definition of J_t, we use $\mathsf{tlocked}_{tl_1, \mathtt{t}, tl_2}(s, n_1, \mathtt{i}, n_2)$ to say that `t` is requesting the lock and its ticket number is `i`. Here tl_1 is the list of the threads which are waiting ahead of `t`, and tl_2 is for the threads behind `t`. Q_t specifies the case when tl_1 is empty. In this case the lock bits must be 0 and tlocked is reduced to lock, as shown at the bottom of Figure 7.18.

The metric $f_\mathsf{t}(\mathfrak{S})$ is determined by the number of threads ahead of `t` in the waiting queue and the status of the lock bits. It decreases

when an environment thread $\mathtt{t}'$ does the definite action $\mathcal{D}_{\mathtt{t}'}$, setting the lock bits to $\mathtt{t}'$. It also decreases when $\mathtt{t}'$ releases the lock and increments `owner`, turning $(\mathtt{L} \neq 0)$ to $(\mathtt{L} = 0)$. Thus we can prove $J \Rightarrow (R, \mathsf{Id}\colon \mathcal{D} \xrightarrow{f} (Q, \mathtt{L{=}0}))$.

By the Soundness Theorem 7.1, we know the ticket lock object satisfies the PSF property, and contextually refines the abstraction generated by the corresponding PSF wrappers, under both strongly and weakly fair scheduling.

7.4.4 Simple PSF Locks with Await Blocks

Figure 7.19 shows the proofs of a simple lock object implemented with an **await** statement which guarantees PSF under weak fairness. The `acquire` method is simply $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\textbf{await}(\mathtt{l{=}0})\{\mathtt{l:=cid}\})$. The `release` method resets the lock bit `l` directly. It has the annotated precondition $(\mathtt{l} = \mathtt{cid})$. We still verify the object in our logic with the specifications `L_ACQ'` and `L_REL` defined in (2.3.2).

We first define the object invariant P used in the OBJ rule. It is defined based on lock, which requires `l` to have the same value as the abstract lock `L`. The queue `listid` records the threads currently waiting for the lock. Here $\mathsf{diff}(tb)$ says that the threads in tb are all different. Then the object invariant $P_{\mathtt{t}}$ further requires that the current thread `t` is not recorded in `listid`. It is preserved before and after `t` calls a method.

The object has three kinds of possible actions (see the definition of G). $\mathit{Req}_{\mathtt{t}}$ appends the thread `t` at the end of `listid` to request the lock (line 1). $\mathit{Acq}_{\mathtt{t}}$ acquires the lock if the lock is available and `t` is at the head of `listid` (lines 2–4). $\mathit{Rel}_{\mathtt{t}}$ releases the lock (line 5). Here $\mathit{Acq}_{\mathtt{t}}$ is also the definite action of thread `t` (see the definition of $\mathcal{D}$). None of the actions are delaying actions.

To verify the **await** statement at lines 2–4, we apply the AWAIT-W rule in Figure 7.14, and prove:

$$\mathsf{lockReq} \;\Rightarrow\; (R\colon \mathcal{D} \circ\!\!\xrightarrow{f} (\mathtt{l} = 0 \wedge \mathtt{cid = enhd(listid)}, \mathtt{L} = 0)). \tag{7.4.1}$$

The metric f is defined at the top of Figure 7.19. We can see that $f_{\mathtt{t}}(\mathfrak{S})$ decreases when an environment thread $\mathtt{t}'$ performs a definite action,

$$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists s, tb.\ \mathsf{lock}_s(tb) \wedge (\mathsf{t} \notin tb) \qquad \text{where } tb ::= \epsilon \mid (\mathsf{t}, \text{`l=0'}) :: tb$$

$$\mathsf{lock}_s(tb) \stackrel{\text{def}}{=} (\mathtt{l} = \mathtt{L} = s) * (\mathtt{listid} = tb) \wedge \mathsf{diff}(tb)$$

$$\mathsf{unlocked}(tb) \stackrel{\text{def}}{=} \mathsf{lock}_0(tb) \qquad \mathsf{lockReq}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists s, tb.\ \mathsf{lock}_s(tb) \wedge (\mathsf{t} \in tb)$$

$$\mathsf{locked}_{\mathsf{t}}(tb) \stackrel{\text{def}}{=} \mathsf{lock}_{\mathsf{t}}(tb) \wedge (\mathsf{t} \neq 0) \wedge (\mathsf{t} \notin tb) \qquad \mathsf{locked}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists tb.\ \mathsf{locked}_{\mathsf{t}}(tb)$$

$$G_{\mathsf{t}} \stackrel{\text{def}}{=} \mathit{Req}_{\mathsf{t}} \vee \mathit{Acq}_{\mathsf{t}} \vee \mathit{Rel}_{\mathsf{t}}$$

$$\mathit{Req}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists s, tb.\ (\mathsf{lock}_s(tb) \wedge (\mathsf{t} \notin tb)) \ltimes \mathsf{lock}_s(tb +\!\!+ [(\mathsf{t}, \text{`l=0'})])$$

$$\mathit{Acq}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists tb.\ \mathsf{unlocked}((\mathsf{t}, \text{`l=0'}) :: tb) \ltimes \mathsf{locked}_{\mathsf{t}}(tb)$$

$$\mathit{Rel}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists tb.\ \mathsf{locked}_{\mathsf{t}}(tb) \ltimes \mathsf{unlocked}(tb)$$

$$\mathcal{D}_{\mathsf{t}} \stackrel{\text{def}}{=} \forall tb.\ \mathsf{unlocked}((\mathsf{t}, \text{`l=0'}) :: tb) \rightsquigarrow \mathsf{locked}_{\mathsf{t}}(tb)$$

$$f_{\mathsf{t}}(\mathfrak{S}) \stackrel{\text{def}}{=} \begin{cases} 2k+1 & \text{if } \exists s, tb, tb'.\ (\mathfrak{S} \models \mathsf{lock}_s(tb +\!\!+ [(\mathsf{t}, \text{`l=0'})] +\!\!+ tb') \\ & \qquad\qquad \wedge\ s \neq 0) \wedge |tb| = k \\ 2k & \text{if } \exists tb, tb'.\ (\mathfrak{S} \models \mathsf{unlocked}(tb +\!\!+ [(\mathsf{t}, \text{`l=0'})] +\!\!+ tb')) \\ & \qquad\qquad \wedge\ |tb| = k \end{cases}$$

```
acquire(){
    {P_cid ∧ arem(L_ACQ')}
 1  listid := listid ++ [(cid, 'l=0')];
    {lockReq_cid ∧ arem(L_ACQ')}
 2  await (l = 0 /\ cid = enhd(listid)) {
      {∃tb. unlocked((cid, 'l=0') :: tb) ∧ arem(L_ACQ')}
 3     l := cid;  listid := listid \ cid;
      {∃tb. locked_cid(tb) ∧ arem(skip)}
 4  }
    {locked_cid ∧ arem(skip)}
}
release(){
    {locked_cid ∧ arem(L_REL)}
 5  l := 0;
    {P_cid ∧ arem(skip)}
}
```

Figure 7.19: Proofs for the simple PSF lock under weak fairness.

since $\mathcal{D}_{\mathsf{t}'}$ will remove t' that is waiting ahead of the thread t. Also $f_{\mathsf{t}}(\mathfrak{S})$ decreases when t' releases the lock, turning $(\mathtt{L} \neq 0)$ to $(\mathtt{L} = 0)$. Thus (7.4.1) holds.

By the Soundness Theorem 7.1, we know this simple lock satisfies PSF under weak fairness.

8

Related Work

There has been much work on the relationships between linearizability, progress properties and contextual refinement (e.g., Filipović *et al.*, 2009; Gotsman and Yang, 2011, 2012; Liang *et al.*, 2013), and on verifying progress properties or progress-aware refinement (e.g., Boström and Müller, 2015; da Rocha Pinto *et al.*, 2016; Gotsman *et al.*, 2009; Hoffmann *et al.*, 2013; Jacobs *et al.*, 2015; Tassarotti *et al.*, 2017). Below we focus on the most closely related work.

8.1 Progress Properties and Abstraction

Herlihy and Shavit (2011) informally discuss the meanings of the progress properties for objects with total methods. Our definitions in Subsection 5.1 mostly follow their explanations, but ours are more formal and close the gap between program semantics and their history-based interpretations. In addition, Herlihy and Shavit (2011) also discuss obstruction-freedom, which we leave as future work.

Filipović *et al.* (2009) first show the equivalence between linearizability and a contextual refinement (which is Theorem 4.1 in this tutorial). Gotsman and Yang (2011) propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom

and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified abstraction theorem to systematically relate all the progress properties plus linearizability to contextual refinements.

Fossati *et al.* (2012) propose a uniform approach in the π-calculus to formulate the standard progress properties for objects with total methods and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated.

There are also formulations of progress properties based on temporal logics. For example, Petrank *et al.* (2009) and Dongol (2006) use linear temporal logics to formalize the progress properties for objects with total methods. Those formulations make it easier to do model checking (e.g., Petrank *et al.*, 2009 also build a tool to model check a variant of lock-freedom), while our abstraction theorem is used to build the program logic LiLi for Hoare-style verification.

In our previous work (Liang *et al.*, 2013), we formulate several contextual refinements, each of which can characterize a progress property of linearizable objects with total methods. However, the contextual refinements lack transitivity because they take different observable behaviors and even assume different scheduling at the concrete and the abstract sides. In this tutorial, we unify WF and LF with the contextual refinement $\sqsubseteq^{\mathsf{t}\omega}$, and unify SF and DF with $\sqsubseteq^{\mathsf{fair}}$ (see Definition 6.1). These contextual refinements have transitivity, so it becomes possible to verify WF/LF (or SF/DF) nested concurrent objects.

8.2 Verification

Using rely-guarantee style logics to verify liveness properties can date back to work by Stark (1985), Stølen (1992), Abadi and Lamport (1995) and Xu *et al.* (1997). Among them the most closely related work is the fair termination rule for while loops proposed by Stølen (1992), based on an idea of wait conditions. His rule requires each iteration to descend if the wait condition P_w holds once in the round. P_w is comparable

to $\neg Q$ in our WHL rule in Figure 7.4. But it is difficult to specify P_w which is part of the global interface of a thread, while our Q can be constructed on-the-fly for each loop. Also it is difficult to construct the well-founded order when $\neg P_w$ is not stable (e.g., as in the TAS lock). We address the problem with the token transfer idea. Besides, his rule does not support starvation-freedom verification.

Gotsman *et al.* (2009) propose a rely-guarantee-style logic to verify non-blocking algorithms. They allow R and G to specify certain types of liveness properties in temporal logic assertions, and do layered proofs iteratively in multiple rounds to break circular reasoning. Afterwards Hoffmann *et al.* (2013) propose the token-transfer idea to handle delays in lock-free algorithms. Their approach can be viewed as giving relatively lightweight guidelines (without the need of multi-round reasoning) to discharge the temporal obligations for lock-freedom verification. We then apply similar ideas in refinement verification (Liang *et al.*, 2014), and the resulting logic can verify linearizability and lock-freedom together. In LiLi, the use of stratified ⧫-tokens generalizes the token-transfer approaches to support delays and rollbacks for deadlock-free objects. Also we propose the new idea of definite actions as a specific guideline to support blocking for progress verification under fair scheduling.

In their program logic Total-TaDA, da Rocha Pinto *et al.* (2016) take a different approach to handle delay. They verify total correctness of *non-blocking* programs by explicitly specifying the number of delaying actions that the environment can do. Recently D'Osualdo *et al.* (2019) propose the program logic TaDA-Live, which extends Total-TaDA with the support for objects with partial methods. We will compare our work with TaDA-Live in detail in Subsection 8.3.

Jacobs *et al.* (2015) also design logic rules for total correctness. They prevent deadlock by global wait orders (proposed by Leino *et al.*, 2010 to prove safety properties), where they need a global function mapping locks to levels. It is unclear if their rules can be applied to algorithms with dynamic locking and rollbacks, such as the list algorithms verified with LiLi. Besides, the idea of wait orders relies on built-in locks, which is ill-suited for object verification since it is often difficult to identify a particular field in the object as a lock.

Boström and Müller (2015) extend the approach of global wait orders to verify finite blocking in non-terminating programs. They propose a notion of obligations which are like our definite actions $\mathcal{D}$. But they still do not support starvation-freedom verification. Here we propose the definite progress condition to also ensure the termination of a thread if it is unblocked infinitely often. On the other hand, they support special built-in blocking primitives for locking, message passing and thread join. Their obligation-based reasoning strategies may be applied to **await** blocks too, to verify that the client threads of **await** will not be permanently blocked.

Schellhorn *et al.* (2016) propose thread-local proof principles for verifying starvation-freedom. They propose a special predicate to describe the waiting-for relations among the threads, and the proof method is based on rely-guarantee and temporal reasoning. In particular, they have a temporal proof obligation saying that not waiting is sufficient for termination. The well-founded metrics we use can be viewed as a way to discharge the temporal proof obligations.

There is also work on verifying refinement under fair scheduling. Back and Xu (1998) and Henzinger *et al.* (2002) propose simulations to verify fair refinement, but their simulations are not thread-local and there is no program logic given. Tassarotti *et al.* (2017) propose a higher-order logic based on Iris (Jung *et al.*, 2015) for fair refinements. Our wrappers and reasoning method may be applied there to support higher-order refinement reasoning with blocking primitives.

None of the above work studies objects with partial methods as we do. On the other hand, our ideas might be general enough to be integrated with these verification methods to support blocking primitives and partial methods.

For objects with partial methods, Gu *et al.* (2016) verify progress of the ticket lock implementation as part of their verified kernel. Their specification of the lock relies on the behaviors of clients. It requires that the client owning a lock must eventually release it. Then they prove that the acquire method always terminates with the cooperative clients. It is unclear how the approach can be applied for general objects with partial methods.

There is also plenty of work for liveness verification based on temporal logics and model checking. Temporal reasoning allows one to verify progress properties in a unified and general way, but it provides less guidance on how to discharge the proof obligations. Our logic rules are based on program structures and enforce specific patterns (e.g., definite actions and tokens) to guide liveness proofs.

8.3 Comparison with TaDA-Live

As we mentioned, TaDA-Live (D'Osualdo *et al.*, 2019) is a program logic for verifying total correctness of client programs that possibly use objects with partial methods. TaDA-Live and our work have several differences.

First, TaDA-Live and LiLi are proposed for different goals. TaDA-Live aims at verifying total correctness, in particular, proving *termination* of programs. It does not care about progress properties of concurrent objects in general. As a result, program specifications in TaDA-Live have to assume cooperative environments, with which the program guarantees to terminate.

By contrast, we mainly focus on specifying and verifying progress properties (not only termination!) of objects, without particular assumptions about how the object methods are invoked by clients. For instance, in PSF/PDF we need to specify the object behaviors even when clients deadlock. Actually most of the technical challenges we address there for partial methods' specification and verification are caused by the weak requirements of clients.

Second, because of the above differences in the verification goals, TaDA-Live and LiLi give different abstractions of objects. TaDA-Live gives *axiomatic* specifications to methods of concurrent objects. The specification is in the form of pre- and post-conditions, and guarantees an abstract atomic view of the method that transforms atomically from a state satisfying the pre-condition to one satisfying the post-condition. As we explain above, TaDA-Live verifies termination of programs and assumes cooperative environments only. The assumption is reflected in the preconditions. For instance, their lock specifications assume in the preconditions that the locks are always eventually released (otherwise

the lock-acquire method may not terminate). For test-and-set locks, their specifications further assume that the lock-acquire/release methods are invoked by the environments for only a bounded number of times. If the preconditions are violated, the specifications become vacuously true and provide no guarantee at all.

In LiLi, we introduce the abstraction theorem in order to describe the abstract object behaviors *operationally*. Since they need to encode various progress behaviors, our abstract specifications for objects may not be atomic (see Section 6). The non-atomicity is a natural consequence of encoding the non-terminating behaviors of objects. Also, as explained above, our specifications do not assume how the object methods are invoked by clients.

As a result, our non-atomic specifications of objects may not be as abstract as those in TaDA-Live. They may cause some complexity in client verification, because we need to replace the method calls with the object specifications and reason about the non-atomic code when verifying clients. The atomic specifications in TaDA-Live are easier to use in verifying (termination of) clients.

On the other hand, our object specifications do not enforce any constraint on the properties of clients that we can verify. In particular, we can verify not only termination, but also non-termination or other liveness properties of non-terminating clients. Besides, our abstraction theorem just decouples the object reasoning from client reasoning, and our operational object specifications do not restrict how to verify the objects or clients. They are not bound with a particular program logic.

Below we give an example to show the difference. It is a solution to the dining philosopher problem of three threads.

```
while (true) {          ||  while (true) {          ||  while (true) {
  lock L1;              ||    lock L2;              ||    lock L1;
  lock L2;              ||    lock L3;              ||    lock L3;
    eat++;              ||      eat++;              ||      eat++;
  unlock L1;            ||    unlock L2;            ||    unlock L1;
  unlock L2;            ||    unlock L3;            ||    unlock L3;
}                       ||  }                       ||  }
```

(8.3.1)

The goal is to prove that `eat` increases infinitely often (then we know the dining philosophers never deadlock). As we mentioned, TaDA-Live's spin lock specification assumes an upper bound on the number of lock-acquire/release operations, which does not hold in (8.3.1), so TaDA-Live cannot apply. But our lock abstractions still apply in this client.

Finally, TaDA-Live achieves better modularity than LiLi because of not only its axiomatic and atomic specifications, but also its use of layered obligation assertions. We use the client below to give a brief overview of TaDA-Live's layered obligation assertions.

```
lock L;                    ||   local d := false;
[done] := true;            ||   while (!d) {
unlock L;                  ||     lock L; d := [done]; unlock L;
                           ||   }
```

(8.3.2)

To verify in TaDA-Live that the client program (8.3.2) must terminate using ticket locks, one needs to introduce two obligation assertions K and D with $\mathsf{layer}(\text{K}) < \mathsf{layer}(\text{D})$. The obligation K says that a thread holding the lock must eventually release the lock. The obligation D is owned by the left thread only. It says that the left thread must eventually set `done` to **`true`** (no matter whether or not it acquires the lock). In a thread's termination proof, one can assume that the environment thread must eventually fulfill its obligation. To avoid circular reasoning, TaDA-Live requires that a thread `t` can only assume the environment thread's obligations with layers strictly lower than any obligation which `t` must fulfill. As a consequence, fulfilling D can rely on the environment thread's obligation K (and indeed it does, since the left thread needs to first acquire the lock before setting `done` to **`true`**, which relies on the right thread releasing the lock), but fulfilling K cannot rely on D.

Below is another example from TaDA-Live (D'Osualdo *et al.*, 2019). The object contains three methods using two ticket locks.

```
both(){                    one(){          two(){
  lock L1; lock L2;          lock L1;        lock L2;
  unlock L2; unlock L1;      unlock L1;      unlock L2;
}                          }               }
```

(8.3.3)

TaDA-Live proves termination of the three methods (in the presence of any concurrent calls of them) by introducing two obligation assertions κ_1 and κ_2, with $\mathsf{layer}(\kappa_2) < \mathsf{layer}(\kappa_1)$. Here κ_1 (or κ_2) says that a thread holding the lock `L1` (or `L2`) must eventually release it. With the layer system, the termination proof can be arranged in a hierarchy.

1. The fulfillment of κ_2 is proved without particular assumptions on the environment's progress. This is like our definite actions, which must be definite regardless of the environment behaviors.

2. The fulfillment of κ_1 is proved by assuming that the environment threads must fulfill κ_2. This assumption is necessary, because in the code `both()`, the thread acquiring `L1` must acquire `L2` before releasing `L1`, and the successful acquirement of `L2` relies on the environment's fulfillment of κ_2.

3. Finally the termination of the three methods are proved by assuming the environment's fulfillment of κ_1 and κ_2.

Our definite actions can be viewed as the most basic obligation assertions which have the lowest layer. That is, the definite actions must be fulfilled by a thread without relying on any progress of its environment. For instance, κ in TaDA-Live's reasoning about (8.3.2) may be defined as a definite action $\mathcal{D}_\kappa$ in LiLi (although LiLi is not designed for verifying client programs, we may still verify (8.3.2) by viewing each thread as an object method). Similarly, to verify the object (8.3.3) in LiLi, we can define a definite action $\mathcal{D}_2$ as the counterpart of the obligation assertion κ_2 in TaDA-Live.

However, LiLi does not support hierarchical definite actions yet, so we cannot directly encode the higher-layer obligation assertions in TaDA-Live, such as δ for (8.3.2) and κ_1 for (8.3.3). Instead, we have to decompose them into definite actions with a finer granularity so that each of them, when enabled, can be fulfilled independently without relying on others. Then our proof of definite progress (see Definition 7.1) in the WHL rule in Figure 7.4 has to determine an order between these flattened definite actions so that each of them is enabled after the fulfillment of those in front of it. Defining these fine-grained definite

actions and finding an order between them may expose details of the program and make the proofs more complicated and less modular.

That said, we could make LiLi more modular by incorporating a similar layer system for definite actions. That is, we may allow definite actions of higher layers, which can be proved "definite" by assuming the fulfillment of those with lower layers. For instance, we may introduce a new definite action $\mathcal{D}_{\text{D}}$ that corresponds to the obligation assertion D in (8.3.2). It has a higher layer, which allows us to prove its fulfillment by assuming $\mathcal{D}_{\text{K}}$. Then in the termination proof of the right thread, we can assume the left thread's fulfillment of both $\mathcal{D}_{\text{D}}$ and $\mathcal{D}_{\text{K}}$. We leave the extension of LiLi with layered definite actions as future work.

9

Conclusion and Future Work

We have studied progress of concurrent objects in three aspects:

- First, we formulate the progress properties. In addition to the traditional progress properties, wait-freedom (WF), lock-freedom (LF), starvation-freedom (SF) and deadlock-freedom (DF), we also introduce two new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF), for concurrent objects with partial methods.

- Second, we design wrappers to generate abstractions for objects. We give distinguished wrappers for different combinations of progress properties and fairness of scheduling. We prove the Abstraction Theorem about the equivalence between each progress property and the progress-aware contextual refinement, where the abstraction is generated by the wrapper.

- Third, we develop a program logic LiLi to verify linearizability and all the six progress properties. We sort progress properties in two dimensions called blocking and delay, use tokens to support delay, and use definite progress conditions to support blocking.

Although our program logic verifies both linearizability and progress properties, it is focused more on the latter. Existing work (Khyzha *et al.*, 2017; Liang and Feng, 2013; Turon *et al.*, 2013b) has shown that linearizability itself can be challenging to verify, and special mechanisms are needed for very fine-grained objects with non-fixed linearization points (LPs). Our logic cannot verify these objects, but our conjecture is that the mechanisms handling non-fixed LPs (as studied in Liang and Feng, 2013) are orthogonal to our progress reasoning, and they can be integrated into our logic if needed.

The key idea of LiLi is to use definite actions and stratified tokens to reason about progress. They can be viewed as special strategies implementing the general principle for termination reasoning, that is to find a well-founded metric that keeps decreasing during the program execution. These ideas and rules give a concrete guide to users on how to construct the metric and the proofs. Although we have tried to make them as general as possible, and they have been shown applicable to many non-trivial algorithms, they may not be complete and it would be unsurprising if there are examples that they cannot handle. As future work, we would like to verify more examples to explore the scope of the applicability.

LiLi is a program logic for *objects*. It verifies linearizability and progress properties of an object. A related project is to develop inference rules to verify liveness properties (e.g., total correctness) of the *client* code. To this end, one way is to use the Abstraction Theorem to replace the concrete object implementations with their abstractions, but we still need to design rules to reason about the abstractions. Since the abstractions are usually non-atomic (see Figures 6.2 and 6.3), the verification may still be not easy. Besides, it is also interesting to extend LiLi to support nested objects and multi-objects. Here a natural question to ask is about the compositionality of progress properties, e.g., whether composing two SF objects gives a SF object.

The specifications of linearizable objects must be *atomic*, but sometimes we may want to give non-atomic specifications to object methods. We can apply our wrappers to every occurrence of the **await** blocks in the non-atomic specifications to establish progress-aware refinements. We suspect that our logic can still be used to verify such refinements

(as in Liang *et al.*, 2014). Another potential limitation may be due to the use of the pure Boolean expression B in **await**$(B)\{C\}$, which may limit the expressiveness of the specifications. However, our technical development does not rely on this setting. Everything may still hold if we replace B with the more expressive state assertions.

We also hope to consider other progress properties. For instance, in addition to WF, LF, SF and DF, there is also an important progress property called obstruction-freedom for objects with total methods in the literature (Herlihy and Shavit, 2008). It guarantees progress for any thread that eventually executes in isolation. It is interesting to design wrappers to generate abstractions for obstruction-free objects, and extend LiLi to also verify obstruction-freedom. Besides progress properties of objects, we would also like to study progress properties of *methods*. For instance, the deadlock-free lazy list algorithm has a "wait-free" `contains` method (Herlihy and Shavit, 2008). How do we specify and verify progress properties of individual methods?

Other interesting future work includes automating the verification process. One of the key problems is to infer the definite actions and prove the definite progress conditions. There have been efforts to synthesize the ranking functions for loop termination (see Cook *et al.*, 2011 for an overview), which may provide insights for automating the definite progress proofs. In addition we might be able to follow the ideas in automated rely-guarantee reasoning (e.g., Calcagno *et al.*, 2007) to automate the verification in our rely-guarantee logic.

Acknowledgements

The material presented here is closely aligned with our work on formalizing and verifying progress of concurrent objects (Liang and Feng, 2016, 2018a; Liang *et al.*, 2013, 2014). This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61922039 and 61632005, and from Huawei Innovation Research Program (HIRP).

References

Abadi, M. and L. Lamport (1995). "Conjoining specifications". *ACM Trans. Program. Lang. Syst.* 17(3): 507–535.

Aspnes, J. and M. Herlihy (1990). "Wait-free data structures in the asynchronous PRAM model". In: *SPAA*. 340–349.

Back, R. and Q. Xu (1998). "Refinement of fair action systems". *Acta Inf.* 35(2): 131–165.

Boström, P. and P. Müller (2015). "Modular verification of finite blocking in non-terminating programs". In: *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 639–663.

Boyapati, C., R. Lee, and M. Rinard (2002). "Ownership types for safe programming: Preventing data races and deadlocks". In: *OOPSLA*. 211–230.

Calcagno, C., M. J. Parkinson, and V. Vafeiadis (2007). "Modular safety checking for fine-grained concurrency". In: *Proceedings of the 14th International Symposium on Static Analysis (SAS 2007)*. 233–248.

Cook, B., A. Podelski, and A. Rybalchenko (2011). "Proving program termination". *Commun. ACM*. 54(5): 88–98.

da Rocha Pinto, P., T. Dinsdale-Young, P. Gardner, and J. Sutherland (2016). "Modular termination verification for non-blocking concurrency". In: *Proceedings of the 25th European Symposium on Programming Languages and Systems (ESOP 2016)*. 176–201.

Derrick, J., G. Schellhorn, and H. Wehrheim (2011). "Mechanically verified proof obligations for linearizability". *ACM Trans. Program. Lang. Syst.* 33(1): 4:1–4:43.

Doherty, S., L. Groves, V. Luchangco, and M. Moir (2004). "Formal verification of a practical lock-free queue algorithm". In: *FORTE*. 97–114.

Dongol, B. (2006). "Formalising progress properties of non-blocking programs". In: *ICFEM*. 284–303.

D'Osualdo, E., A. Farzan, P. Gardner, and J. Sutherland (2019). "TaDA live: Compositional reasoning for termination of fine-grained concurrent programs". arXiv: 1901.05750.

Feng, X. (2009). "Local rely-guarantee reasoning". In: *POPL*. 315–327.

Filipović, I., P. O'Hearn, N. Rinetzky, and H. Yang (2009). "Abstraction for concurrent objects". In: *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. 252–266.

Fossati, L., K. Honda, and N. Yoshida (2012). "Intensional and extensional characterisation of global progress in the π-calculus". In: *CONCUR*. 287–301.

Gotsman, A., B. Cook, M. J. Parkinson, and V. Vafeiadis (2009). "Proving that non-blocking algorithms don't block". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. 16–28.

Gotsman, A. and H. Yang (2011). "Liveness-preserving atomicity abstraction". In: *Proceedings of the 38th International Conference on Automata, Languages and Programming (ICALP 2011)*. 453–465.

Gotsman, A. and H. Yang (2012). "Linearizability with ownership transfer". In: *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. 256–271.

Gu, R., Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo (2016). "CertiKOS: An extensible architecture for building certified concurrent OS kernels". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*. 653–669.

Harris, T. L. (2001). "A pragmatic implementation of non-blocking linked-lists". In: *DISC*. 300–314.

Heller, S., M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit (2005). "A lazy concurrent list-based set algorithm". In: *OPODIS*. 3–16.

Hendler, D., N. Shavit, and L. Yerushalmi (2004). "A scalable lock-free stack algorithm". In: *SPAA*. 206–215.

Henzinger, T. A., O. Kupferman, and S. K. Rajamani (2002). "Fair simulation". *Inf. Comput.* 173(1): 64–81.

Herlihy, M. and N. Shavit (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Herlihy, M. and N. Shavit (2011). "On the nature of progress". In: *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS 2011)*. 313–328.

Herlihy, M. and J. Wing (1990). "Linearizability: A correctness condition for concurrent objects". *ACM Trans. Program. Lang. Syst.* 12(3): 463–492.

Hoffmann, J., M. Marmar, and Z. Shao (2013). "Quantitative reasoning for proving lock-freedom". In: *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*. 124–133.

Jacobs, B., D. Bosnacki, and R. Kuiper (2015). "Modular termination verification". In: *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 664–688.

Jones, C. B. (1983). "Tentative steps toward a development method for interfering programs". *ACM Trans. Program. Lang. Syst.* 5(4): 596–619.

Jung, R., D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer (2015). "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. 637–650.

Khyzha, A., M. Dodds, A. Gotsman, and M. J. Parkinson (2017). "Proving linearizability using partial orders". In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 639–667.

Leino, K. R. M. and P. Müller (2009). "A basis for verifying multi-threaded programs". In: *ESOP*. 378–393.

Leino, K. R. M., P. Müller, and J. Smans (2010). "Deadlock-free channels and locks". In: *ESOP*. 407–426.

Liang, H. and X. Feng (2013). "Modular verification of linearizability with non-fixed linearization points". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 459–470.

Liang, H. and X. Feng (2016). "A program logic for concurrent objects under fair scheduling". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 385–399.

Liang, H. and X. Feng (2018a). "Progress of concurrent objects with partial methods". *Proc. ACM Program. Lang.* 2(POPL): Article 20.

Liang, H. and X. Feng (2018b). "Progress of concurrent objects with partial methods (extended version)". *Tech. Rep.* https://cs.nju.edu.cn/hongjin/papers/popl18-partial-tr.pdf.

Liang, H., X. Feng, and Z. Shao (2014). "Compositional verification of termination-preserving refinement of concurrent programs". In: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. Article 65.

Liang, H., J. Hoffmann, X. Feng, and Z. Shao (2013). "Characterizing progress properties of concurrent objects via contextual refinements". In: *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. 227–241.

Mellor-Crummey, J. M. and M. L. Scott (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Trans. Comput. Syst.* 9(1): 21–65.

Michael, M. M. (2002). "High performance dynamic lock-free hash tables and list-based sets". In: *SPAA*. 73–82.

Michael, M. M. and M. L. Scott (1996). "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *PODC*. 267–275.

Parkinson, M., R. Bornat, and C. Calcagno (2006). "Variables as resource in Hoare logics". In: *LICS*. 137–146.

Petrank, E., M. Musuvathi, and B. Steensgaard (2009). "Progress guarantee for parallel programs via bounded lock-freedom". In: *PLDI*. 144–154.

Schellhorn, G., O. Travkin, and H. Wehrheim (2016). "Towards a thread-local proof technique for starvation freedom". In: *Proceedings of the 12th International Conference on Integrated Formal Methods (IFM 2016)*. 193–209.

Stark, E. W. (1985). "A proof technique for rely/guarantee properties". In: *FSTTCS*. 369–391.

Stølen, K. (1992). "Shared-state design modulo weak and strong process fairness". In: *FORTE*. 479–498.

Tassarotti, J., R. Jung, and R. Harper (2017). "A higher-order logic for concurrent termination-preserving refinement". In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 909–936.

Treiber, R. K. (1986). "System programming: Coping with parallelism". *Tech. Rep.* RJ 5118. IBM Almaden Research Center.

Turon, A., D. Dreyer, and L. Birkedal (2013a). "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency". In: *ICFP*. 377–390.

Turon, A., J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer (2013b). "Logical relations for fine-grained concurrency". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. 343–356.

Vafeiadis, V. (2008). "Modular fine-grained concurrency verification". *Tech. Rep.* PhD Thesis.

Williams, A., W. Thies, and M. D. Ernst (2005). "Static deadlock detection for java libraries". In: *ECOOP*. 602–629.

Xu, Q., W. P. de Roever, and J. He (1997). "The rely-guarantee method for verifying shared variable concurrent programs". *Formal Asp. Comput.* 9(2): 149–174.