A Program Logic for Concurrent Objects under Fair Scheduling (Extended Version)

Hongjin Liang Xinyu Feng

School of Computer Science and Technology & Suzhou Institute for Advanced Study
University of Science and Technology of China
lhj1018@ustc.edu.cn xyfeng@ustc.edu.cn

Abstract

Existing work on verifying concurrent objects is mostly concerned with safety only, e.g., partial correctness or linearizability. Although there has been recent work verifying lock-freedom of non-blocking objects, much less efforts are focused on deadlock-freedom and starvation-freedom, progress properties of blocking objects. These properties are more challenging to verify than lock-freedom because they allow the progress of one thread to depend on the progress of another, assuming fair scheduling.

We propose LiLi, a new rely-guarantee style program logic for verifying linearizability and progress *together* for concurrent objects under fair scheduling. The rely-guarantee style logic unifies thread-modular reasoning about both starvation-freedom and deadlock-freedom in one framework. It also establishes progress-aware abstraction for concurrent objects, which can be applied when verifying safety and liveness of client code. We have successfully applied the logic to verify starvation-freedom or deadlock-freedom of representative algorithms such as ticket locks, queue locks, lock-coupling lists, optimistic lists and lazy lists.

1. Introduction

A concurrent object or library provides a set of methods that allow multiple client threads to manipulate the shared data structure. Blocking synchronization (i.e., mutual exclusion locks), as a straightforward technique to ensure exclusive accesses and to control the interference, has been widely-used in object implementations to achieve linearizability, which ensures the object methods behave as atomic operations in a concurrent setting.

In addition to linearizability, a safety property, object implementations are expected to also satisfy progress properties. The non-blocking progress properties, such as wait-freedom and lock-freedom which have been studied a lot (e.g., [5, 10, 16, 23]), guarantee the termination of the method calls independently of how the threads are scheduled. Unfortunately these properties are too strong to be satisfied by algorithms with blocking synchronization. For clients using lock-based objects, a delay of a thread holding a lock will block other threads requesting the lock. Thus their progress relies on the assumption that every thread holding the lock will eventually be scheduled to release it. This assumption requires *fair* scheduling, i.e., every thread gets eventually executed. As summarized by Herlihy and Shavit [14], there are two desirable progress properties for blocking algorithms, both assuming fair scheduling:

- *Deadlock-freedom*: In each fair execution, there always exists some method call that can finish. It disallows the situation in which multiple threads requesting locks are waiting for each other to release the locks in hand. It ensures the absence of livelock, but not starvation.
- *Starvation-freedom*: Every method call should finish in fair executions. It requires that every thread attempting to acquire a lock should eventually succeed and in the end release the lock. Starvation-freedom is stronger than deadlock-freedom. Nevertheless it can often be achieved by using fair locks [13].

Recent program logics for verifying concurrent objects either prove only linearizability and ignore the issue of termination (e.g., [6, 21, 29, 30]), or aim for non-blocking progress properties (e.g., [5, 10, 16, 23]), 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 Sec. 2.

It is important to note that, although there has been much work on deadlock detection or deadlock-freedom verification (e.g., [4, 20, 31]), deadlock-freedom is often defined as a safety property, which ensures the lack of circular waiting for locks. It does not prevent livelock 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 paper is a liveness property and its definition does not rely on built-in lock primitives. We discuss more related work on liveness verification in Sec. 8.

In this paper we propose LiLi, a new rely-guarantee style logic for concurrent objects under fair scheduling. LiLi is the first program logic that unifies verification of linearizability, starvation-freedom and deadlock-freedom in one framework (the name LiLi stands for Linearizability and Liveness). 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 [24], and is deadlock-free if regular test-and-set (TAS) based spin locks are used. Our work is based on earlier work on concurrency verification, but we make the following new contributions:

- We divide environment interference that affects progress of a thread into two classes, namely *blocking* and *delay*. We show different occurrences of them correspond to the classification of progress into wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom (see Sec. 2.2.1 and Sec. 6). Recognizing the two classes of interference allows us to come up with different mechanisms in our program logic to reason about them separately. Our logic also provides parameterized specifications, which can be instantiated to choose different combinations of the mechanisms. This gives us a unified program logic that can verify different progress properties using the same set of rules.

- We propose two novel mechanisms, *definite actions* and *stratified tokens*, to reason about blocking and delay, respectively. They are also our key techniques to avoid circularity in rely-guarantee style liveness reasoning. A definite action characterizes a thread's progress that does not rely on the progress of the environment. Each blocked thread waits for a queue of definite actions. Starvation-freedom requires the length of the queue be strictly decreasing, while deadlock-freedom allows *disciplined* queue jumps based on the token-transfer ideas [16, 23]. To avoid circular delay, we further generalize the token-transfer ideas by stratifying tokens into multiple levels, which enables us to verify complex algorithms that involve both nested locks and rollbacks (e.g., the optimistic list algorithm).
- By verifying linearizability and progress *together*, we can provide progress-aware abstractions for concurrent objects (see Sec. 5). Our logic is based on termination-preserving simulations as the meta-theory, which establish contextual refinements that assume fair scheduling at both the concrete and the abstract levels. We prove the contextual refinements are equivalent to linearizability and starvation-freedom/deadlock-freedom. The refinements allow us to replace object implementations with progress-aware abstract specifications when the client code is verified. As far as we know, our abstraction for deadlock-free (and linearizable) objects has never been proposed before.
- We have applied our logic to verify simple objects with coarse-grained synchronization using TAS locks, ticket locks [24] and various queue locks (including Anderson array-based locks, CLH locks and MCS locks) [13]. For examples with more permissive locking schemes, we have successfully verified the two-lock queues, and various fine-grained and optimistic list algorithms. To the best of our knowledge, we are the first to formally verify the starvation-freedom/deadlock-freedom of lock-coupling lists, optimistic lists and lazy lists.

Notice that with the assumption of fair scheduling, wait-freedom and lock-freedom are equivalent to starvation-freedom and deadlock-freedom, respectively. Therefore our logic can also be applied to verify wait-free and lock-free algorithms. We discuss this in Sec. 6.

In the rest of this paper, we first analyze the challenges and explain our approach informally in Sec. 2. Then we give the basic technical setting in Sec. 3, and present our logic in Sec. 4, whose soundness theorem, together with the abstraction theorem, is given in Sec. 5. We discuss how our logic supports wait-free and lock-free objects too in Sec. 6. Finally, we summarize the examples we have verified in Sec. 7, and discuss related work and conclude in Sec. 8.

2. Informal Development

Below we first give an overview of the traditional rely-guarantee logic for safety proofs [18], and the way to encode linearizability verification in the logic. Then we explain the challenges and our ideas in supporting liveness verification under fair scheduling.

2.1 Background

Rely-guarantee reasoning. In rely-guarantee reasoning [18], 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 specify the initial and final states respectively.

(a) abstract operation `INC`: `<x++>;`

```
dfInc :
local b := false, r;
while (!b) { b := cas(&L, 0, cid); }  // lock L
r := x;  x := r + 1;  // critical section
L := 0;  // unlock L
```

(b) deadlock-free implementation `dfInc` using a test-and-set lock

```
sfInc :
local i, o, r;
i := getAndInc(next);
o := owner;  while (i != o) { o := owner; }
r := x;  x := r + 1;  // critical section
owner := i + 1;
```

(c) starvation-free implementation `sfInc` using a ticket lock

Figure 1. Counters.

The rely condition R specifies the assumptions on the environment, which are the permitted state transitions that the environment threads may have. The guarantee condition G specifies the possible transitions made by the thread itself. To ensure that parallel threads can collaborate, the guarantee of each thread needs to satisfy the rely of every other thread.

Encoding linearizability verification. As a working example, Fig. 1(b) shows a counter object `dfInc` implemented with a test-and-set (TAS) lock `L`. Verifying linearizability of `dfInc` requires us to prove it has the same abstract behaviors as `INC` in Fig. 1(a), which increments the counter `x` atomically.

Following previous work [21, 23, 30], one can extend a rely-guarantee logic to verify linearizability. We use an assertion $\mathsf{arem}(C)$ to specify as an auxiliary state the abstract operation C to be fulfilled, and logically execute C at the linearization point (LP) of the concrete implementation. For `dfInc`, we prove a judgment in the form of $R, G \vdash \{P \wedge \mathsf{arem}(\texttt{INC})\}\texttt{dfInc}\{Q \wedge \mathsf{arem}(\texttt{skip})\}$. Here R and G specify the object's actions (i.e., lock acquire and release, and the counter updates at both the concrete and the abstract sides) made by the environment and the current thread respectively. P and Q are relational assertions specifying the consistency relation between the program states at the concrete and the abstract sides. The postcondition $\mathsf{arem}(\texttt{skip})$ shows that at the end of `dfInc` there is no abstract operation to fulfill.

2.2 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.

2.2.1 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`.

For a second example, the call of the `dfInc` method (in Fig. 1(b)) by the left thread below may never terminate.

`dfInc(); || while (true) dfInc();`

When the left thread tries to acquire the lock, even if the lock is available at that time, the thread could be preempted by the right thread, who gets the lock ahead of the left. Then the left thread would fail at the `cas` command in the code of `dfInc` and have to loop at least one more round before termination. This may happen infinitely many times, causing non-termination of the `dfInc` method on the left. In this case we say the progress of the left method is *delayed* by its environment's successful acquirement of the lock.

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., acquiring the lock needed by the current thread (even if the lock is subsequently released). 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 `dfInc` example shows, delay from non-terminating clients may cause starvation. Deadlock-free algorithms allow both (with restrictions). As the optimistic list in Fig. 2(a) (explained in Sec. 2.3.4) shows, blocking and delay can be intertwined by the combined use of blocking-based synchronization and optimistic concurrency, which makes the reasoning significantly more challenging than reasoning about lock-free algorithms.

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

2.2.2 Avoid Circular Reasoning

Rely-guarantee style logics provide the power of thread-modular verification by *circular reasoning*. When proving the behaviors of a thread 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 t.

However, circular reasoning is usually unsound in liveness verification [1]. 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. Figure 2(b) 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: 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 `dfInc` example and the optimistic list example. But how do we tell if an action is good or not? The acquirements of locks in Fig. 2(b) do seem to be good because they make the threads progress towards termination. How do we prevent such lock acquirements from delaying others, which may cause circular delay?

2.2.3 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 Sec. 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 `dfInc` in Fig. 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 `sfInc` in Fig. 1(c), the counter becomes starvation-free. Again, there are actually no locks in the programs if we have to work at a low abstraction level to 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., [4, 19]), it is unclear how to apply the techniques for general progress reasoning, with possible combination of locks, ad-hoc synchronization and rollbacks.

2.3 Our Approaches

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 Fig. 2(a). The stratification prevents the circular delay in the example of Fig. 2(b).

Fourth, our delaying actions and definite actions are all semantically specified as part of object specifications, therefore we can support ad-hoc synchronizaiton 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.

2.3.1 Using Tokens to Prevent Infinite Loops

The key to ensuring termination is to require each loop to terminate. Earlier work [16, 23] 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.

2.3.2 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 `sfInc` in Fig. 1(c). It uses the shared variables `owner` and `next` to guarantee the first-come-first-served property of the lock. Initially `owner` equals `next`. To acquire the lock, a thread atomically increments `next` and reads its old value to a variable `i` (line `2`). The value of `i` becomes the thread's ticket. The thread waits until `owner` equals its ticket value `i` (line `3`). Finally the lock is released by incrementing `owner` (line `5`) such that the next waiting thread (the thread with ticket $\mathtt{i}+1$, if there is one) can now enter the critical section.

We can see `sfInc` is not concerned with the circular dependency problem. Intuitively the ticket lock algorithm 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 `t` in this queue, the number of threads ahead of `t` never increases. Thus `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 $P_d \leadsto Q_d$. 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 `sfInc`, the definite action $P_d \leadsto Q_d$ 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 $\mathtt{i}+1$. 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 Sec. 4.2.2.

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 $\mathtt{t}'$) 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 Sec. 2.3.1) and is presented in Sec. 4.2.2. 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 `sfInc`, 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.

2.3.3 Allowing Queue Jumps for Deadlock-Free Objects

The method `dfInc` in Fig. 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, as we explain in Sec. 2.2.1, 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 `dfInc` 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 `dfInc` method. Thus the system as a whole progresses.

Nevertheless, as explained in Sec. 2.2.2, we have to make sure the queue jump (which is a special form of delay) is a "good" one.

We follow the token-transfer ideas [16, 23] to support disciplined queue jumps. We explicitly specify in the rely/guarantee conditions which steps could delay the progress of other threads (jump their queues). To prohibit unlimited queue jumps without making progress, we assign a finite number m of $\blacklozenge$-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 `t`, we require $\mathtt{t}'$ to consume one $\blacklozenge$-token. At the same time, thread `t` could increase $\Diamond$-tokens so that it can loop more rounds. Besides, we 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 $\blacklozenge$-token.

```
 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;
```

(a) optimistic list

```
 1  lock L1;
 2  local r := L2;
 3  while (r != 0) {
 4    unlock L1;
 5    lock L1;
 6    r := L2;
 7  }
 8  lock L2;
 9  unlock L2;
10  unlock L1;
```

(b) rollback

Figure 2. Examples with multiple locks.

2.3.4 Allowing Rollbacks for Optimistic Locking

The ideas we just explained already support simple deadlock-free objects such as `dfInc` in Fig. 1(b), but they cannot be applied to objects with optimistic synchronization, such as optimistic lists [13] and lazy lists [11].

Figure 2(a) shows part of the optimistic list implementation. Each node of the list is associated with a TAS lock, the same lock as in Fig. 1(b). A thread first traverses the list without acquiring any locks (line 3). The traversal `find` returns two adjacent node pointers `p` and `c`. 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 its updates (adding or removing elements) to the list (line 9). Otherwise it releases the two node locks (line 7) and restarts the traversal.

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 Fig. 2(b), as we explain in Sec. 2.2.2.

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 Sec. 2.2.2 with the object in Fig. 2(b) . The deadlock-free optimistic list in Fig. 2(a) 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 Fig. 2(a), 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.

3. Programming Language

Fig. 3 gives the syntax of the language. A program W consists of multiple parallel threads sharing an object Π. We say the threads are the *clients* of the object. An object declaration is a mapping from a method name f to a pair of argument and code (method body). The statements C are similar to those in the simple language

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

Figure 3. Syntax of the programming language.

used for separation logic. The commands $x := [E]$ and $[E] := E'$ reads and writes the memory cell at the location E, respectively. $x := \mathbf{cons}(E, \ldots, E)$ and $\mathbf{dispose}(E)$ allocates and frees memory blocks respectively. The command $\mathbf{print}(E)$ generates externally observable events, which allows us to observe behaviors of programs. We use $\langle C \rangle$ to represent an atomic block in which C is executed atomically. We introduce an auxiliary command **end** to help define liveness properties (see Sec. 5). It is appended at the end of each thread when the program W starts to run, and is not supposed to be used directly by programmers. Executing it generates a special event, which marks the termination of the corresponding thread.

Note that we use one language for both the concrete implementation at the low level and the abstract specification at the high level. We introduce two sets of symbols for each syntactic category (e.g., W and $\mathbb{W}$ for programs, and Π and Γ for objects).

To simplify the presentation, we make several simplifications here. We assume there is only one concurrent object in the system. Each method of the object takes only one argument, and the method body ends with a **return** E command. Also we assume there is no nested method invocation. For the abstract object specification Γ, each method body is an atomic operation $\langle C \rangle$ (ahead of the **return** command), and executing the code is always safe and never blocks.

The model of program states $\mathcal{S}$ is defined in Fig. 4. To ensure that the client code does not touch the object state, in $\mathcal{S}$ we separate the states accessed by clients (σ_c) and by the object (σ_o). Execution of programs is modelled as a labelled transition system $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ in Fig. 5. The event e is defined in Fig. 4. $(\mathtt{t}, \mathbf{obj})$ and $(\mathtt{t}, \mathbf{clt})$ represent an execution step by the thread $\mathtt{t}$ inside and outside the object method, respectively. $(\mathtt{t}, f, n)$ and $(\mathtt{t}, \mathbf{ret}, n)$ record the invocation and the return of an object respectively. The event $(\mathtt{t}, \mathbf{obj}, \mathbf{abort})$ (or $(\mathtt{t}, \mathbf{clt}, \mathbf{abort})$) is generated when the thread $\mathtt{t}$ aborts inside (or outside) the object method body. The output event $(\mathtt{t}, \mathbf{out}, n)$ is generated by the $\mathbf{print}(E)$ command. $(\mathtt{t}, \mathbf{term})$ marks the termination of the thread $\mathtt{t}$, generated by the **end** command. The event $(\mathbf{spawn}, n)$ is used to record the number n of threads in the program, which is helpful to define fair executions (see Sec. 5). We use T to represent an event trace, a (possibly infinite) sequence of events.

Externally observable events. We treat $(\mathtt{t}, \mathbf{out}, n)$, $(\mathtt{t}, \mathbf{obj}, \mathbf{abort})$ and $(\mathtt{t}, \mathbf{clt}, \mathbf{abort})$ as externally observable events, and we use $\mathsf{get\_obsv}(T)$ to represent the subsequence of T consisting of externally observable events only.

4. 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 Fig. 9 and will be explained later.) To verify an object Π, we give a corresponding object specification Γ (see

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

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

(a) program transitions

$$\frac{\Pi(f) = (y, C) \quad [\![E]\!]_{s_c} = n \quad x \in \mathit{dom}(s_c) \quad \kappa = (\{y \leadsto n\}, x, \mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathtt{t}, f, n)}_{\mathtt{t},\Pi} (C, ((s_c, h_c), \sigma_o, \kappa))} \qquad \frac{f \notin \mathit{dom}(\Pi) \quad \text{or} \quad [\![E]\!]_{s_c}\ \mathit{undefined} \quad \text{or} \quad x \notin \mathit{dom}(s_c)}{(\mathbf{E}[\,x := f(E)\,], ((s_c, h_c), \sigma_o, \circ)) \xrightarrow{(\mathtt{t}, \textbf{clt}, \textbf{abort})}_{\mathtt{t},\Pi} \textbf{abort}}$$

$$\frac{\kappa = (s_l, x, C) \quad [\![E]\!]_{s_l} = n \quad s_c' = s_c\{x \leadsto n\}}{(\mathbf{E}[\,\textbf{return}\ E\,], ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathtt{t}, \textbf{ret}, n)}_{\mathtt{t},\Pi} (C, ((s_c', h_c), \sigma_o, \circ))} \qquad \frac{\kappa = (s_l, x, C) \quad [\![E]\!]_{s_l}\ \mathit{undefined}}{(\mathbf{E}[\,\textbf{return}\ E\,], ((s_c, h_c), \sigma_o, \kappa)) \xrightarrow{(\mathtt{t}, \textbf{obj}, \textbf{abort})}_{\mathtt{t},\Pi} \textbf{abort}}$$

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

$$\frac{(C, (s_o \uplus s_l, h_o)) \rightarrowtriangle_{\mathtt{t}} (C', (s_o' \uplus s_l', h_o')) \qquad \mathit{dom}(s_l) = \mathit{dom}(s_l')}{(C, (\sigma_c, (s_o, h_o), (s_l, x, C))) \xrightarrow{(\mathtt{t}, \textbf{obj})}_{\mathtt{t},\Pi} (C', (\sigma_c, (s_o', h_o'), (s_l', x, C)))} \qquad \frac{(C, \sigma_c) \rightarrowtriangle_{\mathtt{t}} (C', \sigma_c')}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathtt{t}, \textbf{clt})}_{\mathtt{t},\Pi} (C', (\sigma_c', \sigma_o, \circ))}$$

(b) thread transitions

$$\frac{(C, \sigma) \rightarrowtriangle_{\mathtt{t}}^{*} (\textbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \rightarrowtriangle_{\mathtt{t}} (\mathbf{E}[\,\textbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \rightarrowtriangle_{\mathtt{t}}^{\omega} \cdot}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \rightarrowtriangle_{\mathtt{t}} (\mathbf{E}[\,\langle C \rangle\,], \sigma)} \qquad \frac{(C, \sigma) \rightarrowtriangle_{\mathtt{t}}^{*} \textbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \rightarrowtriangle_{\mathtt{t}} \textbf{abort}}$$

(c) local thread transitions

Figure 5. Selected operational semantics rules.

$$\begin{array}{rlll}
(\mathit{ThrdID}) & \mathtt{t} & \in & \mathit{Nat} \qquad (\mathit{ExecCtxt}) \quad \mathbf{E} \ ::= \ [\,] \mid \mathbf{E}; C \\
(\mathit{Store}) & s, \mathbb{s} & \in & \mathit{Var} \rightharpoonup \mathit{Int} \qquad (\mathit{Heap}) \quad h, \mathbb{h} \ \in \ \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma, \Sigma & ::= & (s, h) \qquad (\mathit{CallStk}) \quad \kappa, \Bbbk \ ::= \ (s_l, x, C) \mid \circ \\
(\mathit{ThrdPool}) & \mathcal{K}, \mathbb{K} & ::= & \{\mathtt{t}_1 \leadsto \kappa_1, \ldots, \mathtt{t}_n \leadsto \kappa_n\} \\
(\mathit{PState}) & \mathcal{S}, \mathbb{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) \\
(\mathit{Evt}) & e & ::= & (\mathtt{t}, f, n) \mid (\mathtt{t}, \textbf{ret}, n) \mid (\mathtt{t}, \textbf{obj}) \mid (\mathtt{t}, \textbf{clt}) \\
 & & \mid & (\mathtt{t}, \textbf{out}, n) \mid (\mathtt{t}, \textbf{term}) \mid (\textbf{spawn}, n) \\
 & & \mid & (\mathtt{t}, \textbf{obj}, \textbf{abort}) \mid (\mathtt{t}, \textbf{clt}, \textbf{abort}) \\
(\mathit{ETrace}) & T, \mathbb{T} & ::= & \epsilon \mid e :: T \qquad \text{(co-inductive)}
\end{array}$$

Figure 4. States and event traces.

Fig. 3), 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 Sec. 4.1.

Note LiLi is a logic for concurrent objects Π only. We do not provide logic rules for clients. See Sec. 5 for more discussions.

To simplify the presentation in this paper, we describe LiLi based on the plain Rely-Guarantee Logic [18]. Also, to avoid "variables as resources" [26], we assume program variables are either thread-local or read-only. The full version of LiLi (see Appendix A) extends the more advanced Rely-Guarantee-based logic LRG [7] to support dynamic allocation and ownership transfer. It also drops the assumption about program variables.

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

Figure 6. Syntax of the assertion language.

4.1 Assertions

We define assertions in Fig. 6. 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 Fig. 3). 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 Fig. 7.

Figure 7(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 Fig. 7(c)).

Rely/guarantee assertions R and G specify the transitions over the relational states $\mathfrak{S}$. Their semantics is defined in Fig. 7(b). The action $P \ltimes_k Q$ says that the initial relational states satisfy P and the

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

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

(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.

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

$$C \uplus C' \stackrel{\text{def}}{=} \begin{cases} C' & \text{if } C = \textbf{skip} \\ C & \text{if } C' = \textbf{skip} \end{cases}$$

$$(\mathfrak{S}, (u, w), C) \uplus (\mathfrak{S}', (u', w'), C') \stackrel{\text{def}}{=} (\mathfrak{S} \uplus \mathfrak{S}', (u+u', w+w'), C \uplus C')$$

(c) Semantics of full assertions p and q.

Figure 7. Semantics of assertions.

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 Sec. 4.3. $[P]$ specifies identity transitions with the initial states satisfying P. Semantics of $\lfloor G \rfloor_0$ is defined in Sec. 4.3 too (see Fig. 12). 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 Fig. 6, whose semantics is given in Fig. 7(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 $\Diamond$. 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 $\Diamond(E)$ says the number w of $\Diamond$-tokens is *no less than* E. Therefore $\Diamond(0)$ always holds, and $\Diamond(n+1)$ implies $\Diamond(n)$ for any n. We postpone the explanation of u and the assertion $\blacklozenge(E_k, \ldots, E_1)$ to Sec. 4.3. Below we use $\Diamond$ as the shorthand for $\Diamond(1)$. We use $\lfloor p \rfloor_{\Diamond}$ 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 Fig. 7(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):

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

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

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

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

(b) Useful syntactic sugars.

Figure 8. Semantics of definite actions.

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

Definite actions. Fig. 6 also defines definite actions $\mathcal{D}$, whose semantics is given in Fig. 8(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 `x` if `x` is positive initially. It is satisfied by any transitions where initially `x` is not positive. 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 Fig. 8(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 Sec. 4.2.2. Since $\mathcal{D}$ is a special rely/guarantee assertion, the semantics of $\mathcal{D} \Rightarrow \mathcal{D}'$ follows the standard meaning of $R \Rightarrow R'$ [7] (or see the definition in Fig. 12).

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 `L` is acquired by the thread `t`. We use $P_{\mathtt{t}}$ to represent the instantiation of the thread ID parameter in P with `t`, which means P holds on thread `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.

$$\frac{\begin{array}{c}\text{for all } f \in \mathit{dom}(\Pi): \quad \Pi(f) = (x, C) \quad \Gamma(f) = (y, C') \quad \mathit{dom}(\Pi) = \mathit{dom}(\Gamma) \\ \mathcal{D}, R, G \vdash \{P \wedge (x = y) \wedge \mathsf{arem}(C') \wedge \blacklozenge(E_k, \ldots, E_1)\} C \{P \wedge \mathsf{arem}(\mathbf{skip})\} \\ \forall \mathsf{t}, \mathsf{t}'.\, \mathsf{t} \neq \mathsf{t}' \Longrightarrow G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'} \quad \mathsf{wffAct}(R, \mathcal{D}) \quad P \Rightarrow \neg \mathsf{Enabled}(\mathcal{D})\end{array}}{\mathcal{D}, R, G \vdash \{P\} \Pi : \Gamma} \ (\textsc{obj})$$

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

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

$$\frac{p \Rightarrow (E = E') \quad \mathsf{Sta}(p, R)}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbf{return}\ E')\} \mathbf{return}\ E \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbf{skip})\}} \ (\textsc{ret}) \qquad \frac{\mathcal{D}, R, G \vdash \{p\} C_1 \{r\} \quad \mathcal{D}, R, G \vdash \{r\} C_2 \{q\}}{\mathcal{D}, R, G \vdash \{p\} C_1; C_2 \{q\}} \ (\textsc{seq})$$

$$\frac{\mathcal{D}, R, G \vdash \{p \wedge B\} C_1 \{q\} \quad \mathcal{D}, R, G \vdash \{p \wedge \neg B\} C_2 \{q\}}{\mathcal{D}, R, G \vdash \{p\} \mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2 \{q\}} \ (\textsc{if}) \qquad \frac{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_1)\} C \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_2)\}}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_1; C_3)\} C \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(C_2; C_3)\}} \ (\textsc{arem})$$

$$\frac{p' \Rightarrow p \quad R' \Rightarrow R \quad \mathcal{D}, R, G \vdash \{p\} C \{q\} \quad q \Rightarrow q' \quad G \Rightarrow G' \quad \mathsf{Sta}(\{p', q'\}, R) \quad \mathsf{wffAct}(R, \mathcal{D})}{\mathcal{D}, R', G' \vdash \{p'\} C \{q'\}} \ (\textsc{csq})$$

$$\frac{\begin{array}{c}\mathcal{D}, R, G \vdash \{p\} C \{q\} \\ x \notin \mathit{fv}(\mathcal{D}, R, G)\end{array}}{\mathcal{D}, R, G \vdash \{\exists x.\, p\} C \{\exists x.\, q\}} \ (\textsc{ex}) \qquad \frac{\begin{array}{c}\mathcal{D}, R, G \vdash \{p_1\} C \{q_1\} \\ \mathcal{D}, R, G \vdash \{p_2\} C \{q_2\}\end{array}}{\mathcal{D}, R, G \vdash \{p_1 \wedge p_2\} C \{q_1 \wedge q_2\}} \ (\textsc{conj}) \qquad \frac{\begin{array}{c}\mathcal{D}, R, G \vdash \{p_1\} C \{q_1\} \\ \mathcal{D}, R, G \vdash \{p_2\} C \{q_2\}\end{array}}{\mathcal{D}, R, G \vdash \{p_1 \vee p_2\} C \{q_1 \vee q_2\}} \ (\textsc{disj})$$

Figure 9. Inference rules.

4.2 Verifying Starvation-Freedom with Definite Actions

Figure 9 presents the inference rules of LiLi. We explain the logic in two 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 Sec. 4.3.

4.2.1 The OBJ Rule

The OBJ rule requires that each method in Π refine its atomic specification in Γ. Starting from the initial concrete and abstract object states related by 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'. We can temporarily ignore the assertion $\blacklozenge(E_k, \ldots, E_1)$ for deadlock-freedom.

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 reasoning. In Fig. 10 we give a simplified definition of wffAct used in the second premise. Its complete definition is given in Fig. 13, which will be explained later when we introduce stratified actions and $\blacklozenge$-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 $\forall \mathsf{t}'.\, G_{\mathsf{t}'} \Rightarrow [\mathcal{D}_{\mathsf{t}'}] \vee \mathcal{D}_{\mathsf{t}'}$. Therefore, once $\mathcal{D}_{\mathsf{t}}$ is enabled, the only way to disable it is to let the thread t finish the action. As an example, consider the following $\mathcal{D}_{\mathsf{t}}$:

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

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

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.

4.2.2 The WHL Rule for Loops

The WHL rule, shown in Fig. 9, is the most important rule of the logic. It 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 $\lozenge$-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 $\lozenge$-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 $(R, G\colon \mathcal{D}' \xrightarrow{f} Q)$ requires the thread must be waiting for its environment to perform a finite number of definite actions.

Definition 1 (Definite Progress). $\mathfrak{S} \models (R, G : \mathcal{D} \xrightarrow{f} Q)$ 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 $p \Rightarrow J$ and $\mathsf{Sta}(J, R \vee G)$ hold, that is, J is an invariant that holds at every program point of the loop. If $(R, G : \mathcal{D} \xrightarrow{f} Q)$ happens to satisfy the two premises, we can use $(R, G : \mathcal{D} \xrightarrow{f} Q)$ 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 Fig. 10.

Notice in $(R, G : \mathcal{D}' \xrightarrow{f} Q)$ 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 Fig. 8, 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 $\mathcal{D}_1 \wedge \ldots \wedge \mathcal{D}_n$, $\mathcal{D}'$ 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 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 $\Diamond$-tokens and needs to finally terminate, following our argument at the beginning.

4.2.3 More Inference Rules

The HIDE-$\Diamond$ 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 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 Fig. 10. 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

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

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

$p \Rrightarrow q$ iff $\forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F.$
$\quad (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \bot \Sigma_F) \implies \exists C', \Sigma'.$
$\quad\quad ((C, \Sigma \uplus \Sigma_F) \longrightarrow\!\!\triangleright^{*}_{\mathsf{t}} (C', \Sigma' \uplus \Sigma_F)) \wedge ((\sigma, \Sigma'), (u, w), C') \models q$

Figure 10. Auxiliary defs. used in logic rules (simplified version).

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

$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists tl, n_1, n_2.\, \mathsf{lockIrr}_{\mathsf{t}}(tl, n_1, n_2) \qquad G_{\mathsf{t}} \stackrel{\text{def}}{=} \mathit{Lock}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}}$

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

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

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

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

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

$f(\mathfrak{S}) = k$ iff $\mathfrak{S} \models \exists n_1.\, (\mathtt{owner} \mapsto n_1) * \mathsf{true} \wedge (\mathtt{i} - n_1 = k)$

Figure 11. Proofs for the ticket lock (with auxiliary code in gray).

R, we can apply the ATOM-R rule later, which requires that the pre- and post-conditions be stable with respect to R.

4.2.4 Example: Ticket Locks

We prove the starvation-freedom of the ticket lock implementation in Fig. 11 using our logic rules. We have informally discussed in Sec. 2 the verification of the counter using a ticket lock (`sfInc` in Fig. 1(c)). To simplify the presentation, here we erase the increment in the critical section and focus on the progress property of the code in Fig. 11. With an empty critical section, the code functions just as **skip**, so Fig. 11 proves it is linearizable with respect to **skip**. The proofs of `sfInc` (including its starvation-freedom and linearizability with respect to the atomic `INC` in Fig. 1(a)) are given in Appendix C.1.

To help specify the queue of the threads requesting the lock, we introduce an auxiliary array `ticket`. As shown in Fig. 11, each array cell $\mathtt{ticket}[i]$ 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 some predicates to describe the lock status. $\mathsf{lock}(tl, n_1, n_2)$ contains the auxiliary `ticket` array in addition to `owner` and `next`, where $\mathtt{owner} \mapsto n_1$ and $\mathtt{next} \mapsto n_2$, and tl is the list of the threads recorded in $\mathtt{ticket}[n_1], \ldots, \mathtt{ticket}[n_2 - 1]$. We also use $\mathsf{locked}(tl, n_1, n_2)$ for the case when tl is not empty. That is, the lock is acquired by the first thread in tl, while the other threads in tl are waiting for the lock in order. Besides, we use $\mathsf{lockIrr}_{\mathsf{t}}(tl, n_1, n_2)$ 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 does not request the lock. The formal definitions are given in Appendix C.1.

The bottom of Fig. 11 defines the precondition P and the guarantee condition G of the code. G_{t} specifies the possible atomic actions of a thread t. $\mathit{Lock}_{\mathsf{t}}$ adds the thread t at the end of tl of the threads requesting the lock and increments `next`. It corresponds to line 2 of Fig. 11. $\mathit{Unlock}_{\mathsf{t}}$ releases the lock by incrementing `owner` and dequeuing the thread t which currently holds the lock. It corresponds to line 4 of Fig. 11. The rely condition R_{t} includes all the $G_{\mathsf{t}'}$ made by the environment threads t'.

Next we define the definite action $\mathcal{D}$. As we explained in Sec. 2, $\mathcal{D}_{\mathsf{t}}$ requires that whenever the thread t holds a lock with `owner` $\mapsto n_1$, 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 $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ for a stable J. As shown in Fig. 11, we define $J_{\mathtt{t}}$ to say that the thread `t` is requesting the lock. Here $\mathsf{tlocked}_{tl_1,\mathtt{t},tl_2}(n_1, \mathtt{i}, n_2)$ is similar to $\mathsf{locked}(tl_1 :: \mathtt{t} :: tl_2, n_1, n_2)$. It also says that the thread `t` takes the ticket number `i`. $Q_{\mathtt{t}}$ specifies the case when tl_1 is empty (thus `owner` $\mapsto$ `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 `i` and `owner` at that state, which describes the number of threads ahead of `t` in the waiting queue. We use the usual $<$ order on natural numbers as the associated well-founded order. Then, we can verify $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$.

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 Fig. 11.

4.3 Adding Delay for Deadlock-Free Objects

As we explained in Sec. 2, 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 $\blacklozenge$-tokens (we use $\blacklozenge$ here to distinguish them from $\Diamond$-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 Fig. 6, we index the rely/guarantee assertion $P \ltimes_k Q$ with a natural number k and call it a level-k action. We require $0 \le k < \mathsf{maxL}$, 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 Sec. 2.2.2.

To interpret the level numbers in the assertion semantics, we define $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R)$ in Fig. 12 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 Fig. 12.

$\blacklozenge$-tokens in assertions. To ensure the progress of the whole system, we require the steps of delaying actions to pay $\blacklozenge$-tokens. Since we allow multi-levels of transitions to delay other threads, the $\blacklozenge$-tokens are stratified accordingly. Thus we introduce the new assertion $\blacklozenge(E_k, \ldots, E_1)$ in Fig. 6, whose semantics is defined in Fig. 7. It says the number of each level-j $\blacklozenge$-tokens is *no less than* E_j. Here u is a sequence $(n_k, \ldots, n_1)$ recording the numbers of $\blacklozenge$-tokens at different levels, as defined in Fig. 12. We overload $<$ as the dictionary order for the sequence of natural numbers. The ordering over u and other related definitions are also given in Fig. 12.

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

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

$$u ::= (n_k, \ldots, n_1) \quad (1 \le k < \mathsf{maxL})$$

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

$$\begin{array}{l}
(u, w) <_k (u', w') \text{ iff } (u <_k u') \vee (k = 0 \wedge u = u' \wedge w = w') \\
(u, w) \approx_k (u', w') \text{ iff } u \approx_k u' \wedge (k = 0 \Longrightarrow w = w')
\end{array}$$

Figure 12. Levels of state transitions and tokens.

4.3.1 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 $\blacklozenge$-tokens in the precondition of each method.

Below we revisit the inference rules in Fig. 9 and explain their use of multi-level actions and $\blacklozenge$-tokens. In the OBJ rule, we specify in the precondition the number of $\blacklozenge$-tokens needed for each object method. The side condition $\mathsf{wffAct}(R, \mathcal{D})$ is also redefined in Fig. 13, which is explained below.

Decreasing $\blacklozenge$-tokens at the* ATOM *rule. The thread loses $\blacklozenge$-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 Fig. 13. Similar to $p \Rrightarrow q$ in Fig. 10, it allows us to execute the abstract code. Now it also requires the number of $\blacklozenge$-tokens at level k to be decreased if $k \ge 1$.

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 ($k \ge 1$) action from its environment thread `t'`, thread `t` could gain more $\Diamond$-tokens to do more loop iterations. It could also gain more level-k' ($k' < k$) $\blacklozenge$-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 Sec. 2.3.4, the $\blacklozenge$-tokens at all levels in

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

$$\begin{array}{l} p \Rrightarrow_k q \text{ iff } \forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F. \\ \quad (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \bot \Sigma_F) \implies \exists u', w', C', \Sigma'. \\ \quad ((C, \Sigma \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (C', \Sigma' \uplus \Sigma_F)) \wedge (((\sigma, \Sigma'), (u', w'), C') \models q) \\ \quad \wedge\, (u', w') <_k (u, w) \qquad (<_k \text{ defined in Fig. 12}) \end{array}$$

$$\begin{array}{l} \mathsf{Sta}(p, R) \text{ iff } \forall \mathfrak{S}, \mathfrak{S}', u, w, C, k. \\ \quad ((\mathfrak{S}, (u, w), C) \models p) \wedge ((\mathfrak{S}, \mathfrak{S}', k) \models R) \implies \exists u', w'. \\ \quad ((\mathfrak{S}', (u', w'), C) \models p) \wedge ((u', w') \approx_k (u, w)) \\ \qquad\qquad\qquad\qquad\qquad \text{where } \approx_k \text{ is defined in Fig. 12.} \end{array}$$

Figure 13. Key auxiliary definitions for inference rules (final version that supersedes definitions in Fig. 10).

$$\begin{array}{llll} \mathsf{locked}_{\mathsf{t}} & \stackrel{\text{def}}{=} (\mathtt{L} \mapsto \mathsf{t}) & \mathsf{envLocked}_{\mathsf{t}} & \stackrel{\text{def}}{=} \exists \mathsf{t}'.\ \mathsf{locked}_{\mathsf{t}'} \wedge (\mathsf{t}' \neq \mathsf{t}) \\ \mathsf{unlocked} & \stackrel{\text{def}}{=} (\mathtt{L} \mapsto 0) & \mathsf{notOwn}_{\mathsf{t}} & \stackrel{\text{def}}{=} \mathsf{unlocked} \vee \mathsf{envLocked}_{\mathsf{t}} \end{array}$$

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

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

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

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

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

Figure 14. Proofs for the TAS lock.

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 Fig. 13 to reflect the possible increasing of tokens for the thread 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 Sec. 2.3.3, for deadlock-free objects, the environment steps could cause queue jumps to delay the progress of the thread t. Like starvation-free objects, the thread t using deadlock-free objects may wait for a queue of definite actions made by its environment. A queue jump would make the thread t wait for a longer queue of the environment's definite actions.

As shown in Definition 1, the definite progress condition $(R, G : \mathcal{D} \xrightarrow{f} Q)$ 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 $\blacklozenge$-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 Fig. 13. 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 $k \geq 1$.

4.3.2 Example: Test-and-Set Locks

In Fig. 14, we verify the test-and-set lock implementation explained in Sec. 2. Like the ticket lock proofs in Sec. 4.2.4, we simplify the code and prove it is linearizable with respect to **skip**. Here we omit the assertion $\mathsf{arem}(\textbf{skip})$ at each line in the proof, and focus on proving deadlock-freedom of the code.

As defined at the top of Fig. 14, the action $\mathsf{Lock}_{\mathsf{t}}$ (corresponding to the successful `cas` at line 3) is at level 1, which may delay other threads trying to acquire the lock. The $\mathsf{Unlock}_{\mathsf{t}}$ action is at level 0, which 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.

$R, G : \mathcal{D} \xrightarrow{f} Q$ 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 t' cuts in line by acquiring the lock when the lock is free, the current thread t has to wait for $\mathcal{D}_{\mathsf{t}'}$ before t itself progresses. Thus in $R, G : \mathcal{D} \xrightarrow{f} Q$ the current thread `t` can reset its metric $f(\mathfrak{S})$ when its environment acquires the lock.

The detailed proof at the bottom of Fig. 14 shows the changes of tokens. We give the current thread one $\lozenge$-token (using the HIDE-$\lozenge$ rule) to do its loop at lines `2`-`4`. It consumes this $\lozenge$-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 $\lozenge$-token.

Next we stabilize p. For the left branch, if an environment thread t' acquires the lock, which is a delaying action $\mathit{Lock}_{\mathsf{t}'}$, we let the current thread *regain* a $\lozenge$-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 $\lozenge$ to try one more iteration.

4.3.3 Another Example: Nested Locks with Rollback

To demonstrate the use of action levels, we verify the rollback code in Fig. 2(b) which we informally discussed in Sec. 2. Here we assume all the methods of the object either acquire `L1` before `L2` (as in the method of Fig. 2(b)), or acquire only one lock.

Stratified delaying actions. As in the TAS lock example in Sec. 4.3.2, 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, defined as *Lock2* in Fig. 15, 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 t' before the current thread `t` starts the method. Then `t` would continuously roll back until 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 $\blacklozenge$-tokens infinitely often, and consequently require an infinite number of $\blacklozenge$-tokens be assigned to the method at the beginning, which is impossible. To address the problem, we define in Fig. 15 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.

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

Figure 15. Multi-level actions for the example in Fig. 2(b).

```
   { $\mathsf{notOwn_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2) \wedge \blacklozenge(1,1)$ }
1  lock L1;
   $p_1 \overset{\text{def}}{=}$ { $\mathsf{locked_{cid}}(L1) * (\mathsf{unlocked}(L2) \wedge \blacklozenge(1,0)$
                          $\vee\ \mathsf{envLocked_{cid}}(L2) \wedge \blacklozenge(1,1))$ }
2  local r := L2;
   { $\mathsf{locked_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2)$
     $\wedge ((r = 0) \wedge \blacklozenge(1,0)\ \vee\ (r \neq 0) \wedge \blacklozenge(1,1) \wedge \lozenge)$ }
3  while (r != 0) {
     $p_2 \overset{\text{def}}{=}$ { $\mathsf{locked_{cid}}(L1)$
            $* (\mathsf{unlocked}(L2) \vee \mathsf{envLocked_{cid}}(L2) \wedge \lozenge) \wedge \blacklozenge(1,1)$ }
4    unlock L1;
5    lock L1;
     { $\mathsf{locked_{cid}}(L1) * (\mathsf{unlocked}(L2) \wedge \blacklozenge(1,0)$
                     $\vee\ \mathsf{envLocked_{cid}}(L2) \wedge \blacklozenge(1,1) \wedge \lozenge)$ }
6    r := L2;
     { $\mathsf{locked_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2)$
       $\wedge ((r = 0) \wedge \blacklozenge(1,0)\ \vee\ (r = 1) \wedge \blacklozenge(1,1) \wedge \lozenge)$ }
7  }
   { $\mathsf{locked_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2) \wedge \blacklozenge(1,0)$ }
8  lock L2;
   { $\mathsf{locked_{cid}}(L1) * \mathsf{locked_{cid}}(L2)$ }
9  unlock L2;
10 unlock L1;
   { $\mathsf{notOwn_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2)$ }

 $Q_t \overset{\text{def}}{=} (\mathsf{locked_t}(L1) \vee \mathsf{unlocked}(L1)) * \mathsf{unlocked}(L2)$
```

Figure 16. Proof outline for the rollback example in Fig. 2(b).

To simplify the presentation, the definitions in Fig. 15 follow the notations in LRG [7], 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 Fig. 15, $\mathcal{D}2$ says a thread 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 Fig. 16, 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 Fig. 14.

`lock L1` at line `1` is implemented using the TAS lock, and its detailed proof is in Fig. 17, which will be explained later. The acquirement of `L1` may or may not consume a level-1 $\blacklozenge$-token, depending on whether `L2` is free or not (see p_1 in Fig. 16). 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 Fig. 16. 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 Fig. 16, 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 $\lozenge$-token for the loop (applying the HIDE-$\lozenge$ 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 $\lozenge$-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 $\lozenge$-token.

Besides, the definite progress $R, G\colon \mathcal{D} \xrightarrow{f} Q$ 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:

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

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`.

```
 $p_{01} \overset{\text{def}}{=} \mathsf{unlocked}(L1)$            $* \mathsf{unlocked}(L2)$
 $p_{02} \overset{\text{def}}{=} (\mathsf{unlocked}(L1)$         $* \mathsf{envLocked_{cid}}(L2)) \wedge \lozenge$
 $p_{03} \overset{\text{def}}{=} (\mathsf{envLocked_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2))$     $\wedge \lozenge$
   { $\mathsf{notOwn_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2) \wedge \blacklozenge(1,1)$ }
1  local b := false;
   { $(\neg b) \wedge (\mathsf{notOwn_{cid}}(L1) * \mathsf{notOwn_{cid}}(L2)) \wedge \blacklozenge(1,1) \wedge \lozenge$
     $\vee\ b \wedge p_1$ }
2  while (!b) {
     { $(p_{01} \vee p_{02} \vee p_{03}) \wedge \blacklozenge(1,1)$ }
3    b := cas(L1, 0, cid);
     { $b \wedge \mathsf{locked_{cid}}(L1)$
       $* (\mathsf{unlocked}(L2) \wedge \blacklozenge(1,0) \vee \mathsf{envLocked_{cid}}(L2) \wedge \blacklozenge(1,1))$
       $\vee (\neg b) \wedge ((\mathsf{unlocked}(L1) \vee \mathsf{envLocked_{cid}}(L1))$
                 $* \mathsf{notOwn_{cid}}(L2)) \wedge \blacklozenge(1,1) \wedge \lozenge$ }
4  }
   { $p_1$ }
```

Figure 17. Proof outline for `lock L1` in the rollback example.

Acquirements of `L1` in the rollback example. Finally we discuss the proof of the implementation of `lock L1` in Fig. 17. Here we use the same Q in Fig. 16 to verify the loop. That is, we think the thread is "blocked" if `L2` is not available. Initially we give one $\lozenge$-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 $\lozenge$-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 $\lozenge$-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 Fig. 16. 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 $\lozenge$-token for the next round of loop.

It may seem strange that for the loop we do not use a $Q' \overset{\text{def}}{=} \mathsf{locked}_{\mathtt{t}}(\mathtt{L1}) \vee \mathsf{unlocked}(\mathtt{L1})$, i.e., the same Q as in Fig. 14. 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 $p'_{02} \overset{\text{def}}{=} \mathsf{envLocked}_{\mathtt{cid}}(\mathtt{L1}) * \mathsf{notOwn}_{\mathtt{cid}}(\mathtt{L2})$. 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.

5. Soundness and Abstraction Theorems

Our logic LiLi is a sound proof technique for concurrent objects based on blocking algorithms, as shown by the following theorem.

Theorem 2 (Soundness)**.** If $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$, then

(1) both $\Pi \preceq^{\mathsf{lin}}_P \Gamma$ and $\mathsf{deadlock\text{-}free}_P(\Pi)$ hold; and
(2) if $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\mathsf{starvation\text{-}free}_P(\Pi)$ holds.

Here $\Pi \preceq^{\mathsf{lin}}_P \Gamma$ describes linearizability of the object Π. Informally it says that Π has the same effects as the *atomic* operations of Γ (assuming the initial object states satisfy P). The formal definition is standard [15] and omitted here. $\mathsf{deadlock\text{-}free}_P(\Pi)$ and $\mathsf{starvation\text{-}free}_P(\Pi)$ are the two progress properties defined following their informal meanings [14] (or see Sec. 1).

Theorem 2 shows that LiLi ensures linearizability and deadlock-freedom *together*, and it also ensures starvation-freedom when the rely/guarantee specification of the object satisfies certain constraints. The constraints $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$ require R and G to specify actions of level 0 only. 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 starvation-freedom.

Abstraction. The soundness theorem shows that our logic ensures linearizability with respect to atomic operations. However, from the client code's point of view, the methods of deadlock-free objects do *not* refine atomic operations when termination is concerned. Consider the example below.

```
dfInc(); s:=1; || while (s=0) dfInc();
    INC; s:=1; || while (s=0) INC;
```

The first line shows the client code using a lock-free counter, while the second line uses an atomic counter (see Fig. 1 for the implementation of counters). Assuming s=0 initially, it is easy to see the first program may or may not terminate, but the second one must terminate under *fair scheduling*. Therefore the first program is *not* a termination-preserving refinement of the second one. Note that if we replace dfInc with the starvation-free counter sfInc, the first program must terminate too under fair scheduling.

We propose a novel "progress-aware" object specification $\mathsf{wr}_1(\Gamma)$ for deadlock-free objects that are linearizable with respect to Γ. As defined below, $\mathsf{wr}_1(\Gamma)$ wraps the atomic operations in Γ with some auxiliary code for synchronization.

$\mathsf{wr}_1(\Gamma)(f) \overset{\text{def}}{=} (x, \mathsf{wr}_1(\langle C \rangle); \textbf{return } E)$ if $\Gamma(f) = (x, \langle C \rangle; \textbf{return } E)$

```
wr1(⟨C⟩) ≝ local u1 := nondet(), u2 := nondet();
           while(u1 >= 0) { lock l; unlock l; u1--; }
           ⟨C⟩;
           while(u2 >= 0) { lock l; unlock l; u2--; }
```

Here we assume l is a fresh variable, i.e., $\mathtt{l} \notin \mathit{fv}(\Pi, \Gamma, P)$. The wrapper function wr_1 inserts a finite (but arbitrary) number of lock-acquire (lock l) and lock-release (unlock l) actions before and after the atomic abstract code $\langle C \rangle$. The command u := nondet() assigns to u a nondeterministic number. The lock l is a TAS lock (implementation shown in Fig. 1(b)). Then the progress of a thread executing $\mathsf{wr}_1(\Gamma)$ could be delayed by other threads acquiring the lock l. By introducing the explicit delay mechanism, the abstract specification can model the deadlock-freedom property. In our previous example, if we replace INC with $\mathsf{wr}_1(\mathtt{INC})$, the second program may fail to terminate too, even under fair scheduling.

Before showing our abstraction theorem, we first define contextual refinement below.

Definition 3 (Contextual Refinement under Fair Scheduling)**.**

$$\begin{array}{l}\Pi \sqsubseteq_P \Pi' \;\text{ iff }\; \forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \Sigma_o.\; ((\sigma_o, \Sigma_o) \models P) \\ \qquad\qquad \Longrightarrow \mathcal{O}_{f\omega}[\![(W, (\sigma_c, \sigma_o))]\!] \subseteq \mathcal{O}_{f\omega}[\![(W', (\sigma_c, \Sigma_o)]\!]\,; \\ \text{where } W = \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \text{ and } W' = \textbf{let } \Pi' \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n.\end{array}$$

Here $\mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ generates the full traces of externally observable events in *fair* executions starting from $(W, \mathcal{S})$. In our language in Sec. 3, only outputs (produced by **print** commands) and fault events are externally observable. Note that $\mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ contains only *full* execution traces (which could be infinite for non-terminating executions), therefore $\sqsubseteq$ is a termination-preserving refinement.

As an important and novel result, we show the following abstraction theorem, saying that our logic LiLi ensures the contextual refinements $\sqsubseteq$.

Theorem 4 (Progress-Aware Abstraction)**.**
Suppose $\mathtt{l} \notin \mathit{fv}(\Pi, \Gamma, P, \mathcal{D}, R, G)$. If $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$, then

(1) $\Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma)$ holds; and
(2) if $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\Pi \sqsubseteq_P \Gamma$ holds.

where $\mathsf{wr}_1(P)$ extends the precondition P with the lock variable l, i.e., $\mathsf{wr}_1(P) \overset{\text{def}}{=} (P * (\mathtt{l} \mapsto 0))$.

The contextual refinements $\sqsubseteq$ provide abstractions for concurrent objects under fair scheduling, which can be applied for modular verification of client code. When proving liveness properties of a client of an object under fair scheduling, we can soundly replace the concrete object implementation Π by some more abstract code. For starvation-free objects (case (2) in the theorem), the substitute is Γ, the atomic abstract operations. For deadlock-free objects (case (1) in the theorem), the substitute is $\mathsf{wr}_1(\Gamma)$, where the atomic abstract operations Γ are wrapped with auxiliary code for synchronization. In this paper we do not discuss the verification of clients.

More details about proofs. Due to splace limit, we omit definitions of some key concepts, e.g., linearizability, deadlock-freedom and starvation-freedom. They are mostly standard and are presented in Appendix B.5. The proofs of Theorems 2 and 4 are shown in detail in Appendix B. Here we only give a brief overview about the structure of the proofs.

To prove Theorem 4, we introduce a termination-preserving simulation, which extends previous work [23] to reason about blocking and delay. LiLi ensures that the concrete implementation Π is simulated by the abstract specification (Γ for starvation-free objects and $\mathsf{wr}_1(\Gamma)$ for deadlock-free ones). Then we prove the simulation ensures the contextual refinement $\sqsubseteq$.

We also establish the equivalence between the contextual refinements and the combination of linearizability and deadlock-freedom/starvation-freedom, as in Liang et al.'s previous work [22]. Then Theorem 2 follows from these equivalence results and Theorem 4.

In more detail, Theorem 2 is proved in a unified framework in Fig. 18. In the diagram, we use "$\Longrightarrow$" for implications and "$\Longleftrightarrow$" for equivalences. The dashed arrow $a \dashrightarrow b$ and the dotted arrow $b \cdots\!\!\rightarrow a$ both mean that b is an instance of a.

The top at Fig. 18 shows our final goals: verifying linearizability $\Pi \preceq^{\mathsf{lin}}_P \Gamma$, and $\mathsf{starvation\text{-}free}_P(\Pi)$ or $\mathsf{deadlock\text{-}free}_P(\Pi)$. Inspired by earlier work [8, 22], we reduce the goals (see ① and ② in Fig. 18) to verifying contextual refinements $\sqsubseteq$ in Definition 3.

$$\begin{tikzcd}
\Pi \preceq^{\mathsf{lin}}_P \Gamma \wedge \mathsf{starvation\text{-}free}_P(\Pi) \arrow[d, Leftrightarrow, "①"'] & & \Pi \preceq^{\mathsf{lin}}_P \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \arrow[d, Leftrightarrow, "②"] \\
\Pi \sqsubseteq_P \Gamma & & \Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma) \\
& \Pi \sqsubseteq_{P'} \Pi' \arrow[ul, dashed, "③"] \arrow[ur, dashed, "④"'] & \\
& \Pi \precsim_{P'} \Pi' \arrow[u, Rightarrow] & \\
\Pi \precsim_P \Gamma \arrow[ur, dotted, "⑤"] & & \Pi \precsim_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma) \arrow[ul, dotted, "⑥"'] \\
& \mathcal{D}, R, G \models \{P\}\Pi : \Gamma \arrow[ul, Rightarrow, "⑦"', "{(R \Rightarrow \lfloor R \rfloor_0) \wedge (G \Rightarrow \lfloor G \rfloor_0)}"] \arrow[ur, Rightarrow, "⑧"'] & \\
& \mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma \arrow[u, Rightarrow, "⑨"] &
\end{tikzcd}$$

Figure 18. A unified framework for logic soundness proofs.

When using $\sqsubseteq$ to characterize deadlock-free objects Π (see ② in Fig. 18), their methods at the abstract side are no longer atomic. We need to use the "progress-aware" object specification $\mathsf{wr}_1(\Gamma)$.

Our results ① and ② in Fig. 18 unify starvation-freedom and deadlock-freedom. Both can be characterized by contextual refinement $\Pi \sqsubseteq_{P'} \Pi'$, where Π' may not be atomic. (See ③ and ④.)

Next we propose the simulation $\Pi \precsim_{P'} \Pi'$ as a proof technique for the contextual refinement $\sqsubseteq$. (See the gray box at the center of Fig. 18.) The simulation ensures that executions of Π preserve the behaviors of Π' under fair scheduling. It is adapted from the compositional simulations in earlier work [21, 23].

At the bottom ⑨ of Fig. 18, we define semantics for our logic judgment $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$. Note that the logic uses the atomic Γ as specification. The judgment semantics $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ is also based on a simulation, but it is much closer to the logic rules, which can simplify the task of proving the validity of each rule in Fig. 9. It can be directly (see ⑧) translated to $\Pi \precsim_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma)$, where wr_1 is the wrapper function we just defined (still assume `l` is an arbitrary fresh variable). If the rely/guarantee conditions R and G specify actions of level 0 only (see case (2) in Theorem 2), the judgment semantics can also be reduced to $\Pi \precsim_P \Gamma$ (see ⑦). Both $\Pi \precsim_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma)$ and $\Pi \precsim_P \Gamma$ are instances of the more general simulation $\Pi \precsim_{P'} \Pi'$ (see ⑤ and ⑥).

The proof framework in Fig. 18 unifies the logic soundness proofs for both starvation-freedom and deadlock-freedom. All the formal definitions and detailed proofs are given in Appendix B.

6. On Lock-Freedom and Wait-Freedom

As a program logic for concurrent objects under fair scheduling, LiLi unifies the verification of linearizability, starvation-freedom and deadlock-freedom. It has been applied to verify objects with blocking synchronization (i.e., mutual exclusion locks).

LiLi can also be applied to verify non-blocking objects. For non-blocking objects, wait-freedom and lock-freedom are two commonly accepted progress criteria, which require method-wise progress and whole-system progress respectively. Then, under fair scheduling, wait-freedom and lock-freedom are degraded to starvation-freedom and deadlock-freedom, respectively.

Fig. 19 shows the relationships among all the four progress properties (where "$\Rightarrow$" represents implications). We sort them in two dimensions: *blocking* and *delay* (their difference has been explained in Sec. 2.2.1). Starvation-free or deadlock-free objects allow a thread to be blocked, and lock-free and deadlock-free objects permit delay.

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

Figure 19. Progress properties of concurrent objects.

Our logic LiLi handles blocking by definite actions, and supports delay by $\blacklozenge$-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 Fig. 19.

To verify lock-free objects, we instantiate the definite actions $\mathcal{D}$ to be $\mathsf{false} \leadsto \mathsf{true}$, 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 Fig. 9 is reduced to requiring that the loop terminates (the $\lozenge$-tokens decrease at each iteration) unless being delayed by the environment. The definite progress condition $J \Rightarrow (R, G\colon \mathcal{D} \xrightarrow{f} Q)$ could trivially hold by setting both Q and J to be `true` and f to be a constant function.

To verify wait-free objects, besides instantiating $\mathcal{D}$ as $\mathsf{false} \leadsto \mathsf{true}$, we also require R and G to specify actions of level 0 only, as in Theorem 2(2). The instantiation results in the logic rules disallowing both blocking and delay, so we know every method would terminate regardless of the environment interference.

In fact, Liang et al.'s program logic rules [23] for lock-free algorithms can be viewed as a specialization of LiLi. Thus all the examples verified in their work can also be verified in LiLi.

7. More Examples

We have seen a few small examples showing the use of LiLi. Below we give an overview of other blocking algorithms we have verified. Their proofs are in Appendix C (for starvation-free examples) and Appendix D (for deadlock-free examples).

- *Coarse-grained synchronization.* The easiest way to implement a concurrent object is using a single lock to protect all the object data. Our logic can be applied to such an object. As an example, we verified the counter with various lock implementations [13, 24], 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 verified Michael-Scott two-lock queues [25], lock-coupling lists [13], optimistic lists [13], and lazy lists [11]. 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.

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

Optimistic lists. Below we verify the optimistic list-based implementation in Fig. 2(a) 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. Fig. 20 shows the code and the proof outline for `rmv`.

We have informally explained the idea of the algorithm in Sec. 2.3.4. To verify its progress in LiLi, we need to recognize the delaying actions, specify them in rely and guarantee conditions

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

Figure 20. Proofs for optimistic lists (with auxiliary code in gray).

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 [30], 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 Sec. 4.3.2. 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.

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 Fig. 20, the precondition is given $♦(1,2)$, two level-1 ♦-tokens for locking two adjacent nodes, and one level-2 ♦-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 $\Diamond$-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 $\Diamond$-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). The full specifications and proofs are given in Appendix D.4.

8. Related Work and Conclusion

Using rely-guarantee style logics to verify liveness properties can date back to work by Stark [27], Stølen [28], Abadi and Lamport [1] and Xu et al. [32]. Among them the most closely related work is the fair termination rule for while loops proposed by Stølen [28], 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 Fig. 9. 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. [10] 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. [16] 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. Liang et al. [23] then apply similar ideas in refinement verification. Their logic can verify linearizability and lock-freedom together. In LiLi, the use of stratified ♦-tokens generalizes their 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.

Recently, da Rocha Pinto et al. [5] 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. As we explained, blocking and delay are two different kinds of interference causing non-termination, both of which are now handled in LiLi.

Jacobs et al. [17] also design logic rules for total correctness. They prevent deadlock by global wait orders (proposed by Leino et al. [20] 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 [3] 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.

Filipović et al. [8] first show the equivalence between linearizability and a contextual refinement. Gotsman and Yang [9] suggest a connection between lock-freedom and a termination-sensitive contextual refinement. Afterwards Liang et al. [22] formulate several contextual refinements, each of which can characterize a liveness property of linearizable objects. However, their contextual refinements for blocking properties assume fair scheduling at the concrete level only, which lack transitivity. In this paper, we unify deadlock-freedom and starvation-freedom with the contextual refinement $\sqsubseteq$ (see Def. 3) which gives us the novel Abstraction Theorem (Thm. 4) to support modular reasoning about client code.

Back and Xu [2] and Henzinger et al. [12] propose simulations to verify refinement under fair scheduling. Their simulations are not thread-local, and there is no program logic given.

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.

Conclusion and future work. We propose LiLi to verify linearizability and starvation-freedom/deadlock-freedom of concurrent objects. It is the first program logic that supports progress verification of blocking algorithms. We have applied it to verify several non-trivial algorithms, including lock-coupling lists, optimistic lists and lazy lists. In the future, we would like to further test its applicability with more examples, such as tree algorithms which perform rotation by fine-grained locking. We also hope to mechanize LiLi and develop tools to automate the verification process.

Acknowledgments

We would like to thank Zhong Shao and anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61229201, 61379039, 91318301 and 61502442.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 1995.

[2] R. Back and Q. Xu. Refinement of fair action systems. *Acta Inf.*, 1998.

[3] P. Boström and P. Müller. Modular verification of finite blocking in non-terminating programs. In *ECOOP*, pages 639–663, 2015.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[5] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular termination verification for non-blocking concurrency, 2015. Manuscript.

[6] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *TOPLAS*, 2011.

[7] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[8] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 2010.

[9] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, pages 453–465, 2011.

[10] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.

[11] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS'05*.

[12] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. *Inf. Comput.*, 2002.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[14] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[15] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 1990.

[16] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013.

[17] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular termination verification. In *ECOOP*, pages 664–688, 2015.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.

[19] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP 2009*, pages 378–393, 2009.

[20] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, pages 407–426, 2010.

[21] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[22] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.

[23] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, 2014.

[24] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 1991.

[25] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[26] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146, 2006.

[27] E. W. Stark. A proof technique for rely/guarantee properties. In *FSTTCS*, pages 369–391, 1985.

[28] K. Stølen. Shared-state design modulo weak and strong process fairness. In *FORTE*, 1992.

[29] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[30] V. Vafeiadis. Modular fine-grained concurrency verification, 2008. PhD Thesis.

[31] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP*, pages 602–629, 2005.

[32] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

A. The LRG-Style Full Version of LiLi

The full version of LiLi extends the advanced Rely-Guarantee-based logic LRG [7] to support dynamic allocation and ownership transfer. The top level judgment is now in the form of $\mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma$. Here the fence I is used to determine the boundary of the shared memory following LRG [7]. Just like P, I is also a relational assertion specifying the consistency relation between the concrete data representation and the abstract value.

A.1 LRG-Style Assertions

$$\begin{array}{rrl} (\mathit{RelAssn}) & P, Q, I, J & ::= \ \ldots \ \mid \ \mathsf{own}(x) \\ (\mathit{RelAct}) & R, G & ::= \ \ldots \ \mid \ G * G \end{array}$$

Following LRG [7], we treat program variables as resources [26] and use $\mathsf{own}(x)$ for the ownership of the program variable x. Also we introduce separating conjunction over actions to locally define shared state updates. $G_1 * G_2$ means that the actions G_1 and G_2 start from disjoint relational states and the resulting states are also disjoint. Here a level-k transition $(\mathfrak{S}, \mathfrak{S}')$ can be relaxed to a transition at a level $k' \leq k$. Their semantics is defined below.

$$(\sigma, \Sigma) \uplus (\sigma', \Sigma') \stackrel{\text{def}}{=} (\sigma \uplus \sigma', \Sigma \uplus \Sigma') \ \text{ where } \ (s, h) \uplus (s', h') \stackrel{\text{def}}{=} (s \uplus s', h \uplus h')$$

$$\begin{array}{lll} ((s, h), (\mathbb{s}, \mathbb{h})) \models \mathsf{own}(x) & \text{iff} & \mathit{dom}(s \uplus \mathbb{s}) = \{x\} \\ (\mathfrak{S}, \mathfrak{S}') \models G_1 * G_2 & \text{iff} & \exists \mathfrak{S}_1, \mathfrak{S}_2, \mathfrak{S}'_1, \mathfrak{S}'_2.\, \mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2 \wedge \mathfrak{S}' = \mathfrak{S}'_1 \uplus \mathfrak{S}'_2 \\ & & \qquad \wedge ((\mathfrak{S}_1, \mathfrak{S}'_1) \models G_1) \wedge ((\mathfrak{S}_2, \mathfrak{S}'_2) \models G_2) \\ \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), G_1 * G_2) & \stackrel{\text{def}}{=} & \mathit{min}\{k \mid \exists \mathfrak{S}_1, \mathfrak{S}_2, \mathfrak{S}'_1, \mathfrak{S}'_2.\, (\mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2) \wedge (\mathfrak{S}' = \mathfrak{S}'_1 \uplus \mathfrak{S}'_2) \\ & & \qquad \wedge\, k = \mathit{max}(\mathcal{L}((\mathfrak{S}_1, \mathfrak{S}'_1), G_1), \mathcal{L}((\mathfrak{S}_2, \mathfrak{S}'_2), G_2))\} \\ R \Rightarrow R' & \text{iff} & \forall \mathfrak{S}, \mathfrak{S}', k.\, ((\mathfrak{S}, \mathfrak{S}', k) \models R) \Longrightarrow \exists k' \leq k.\, (\mathfrak{S}, \mathfrak{S}', k') \models R' \end{array}$$

The syntactic sugars Id, Emp and True represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

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

Fence. Since we logically split states into local and shared parts as in LRG [7], we need a precise invariant I to uniquely determine the boundary between local and shared resources. We define the fence $I \triangleright G$ below, which says that the transition G must be made within the boundary specified by I.

$$I \triangleright G \ \text{ iff } \ ([I] \Rightarrow G) \wedge (G \Rightarrow (I \ltimes I)) \wedge \mathsf{Precise}(I)$$

The formal definition of the precise requirement $\mathsf{Precise}(I)$ is given in earlier work [21, 23], which follows its usual meaning as in separation logic but is now interpreted over relational states. Since the need of fenced rely/guarantee conditions is inherited from LRG and is orthogonal to the problem we study in this paper, readers unfamiliar with LRG can safely ignore it.

A.2 Inference Rules

Figure 21 presents the complete set of inference rules of LiLi.

Executing abstract code at (ATOM) and (CSQ) rules. Below we first define $p \stackrel{G}{\Rrightarrow} q$ used in the (CSQ) rule.

$$\begin{array}{l} p \stackrel{G}{\Rrightarrow} q \ \text{ iff } \ \forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F. \\ \quad (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \bot \Sigma_F) \implies \exists k, u', w', C', \Sigma'. \\ \quad ((C, \Sigma \uplus \Sigma_F) \longrightarrow\!\!\triangleright^*_{\mathsf{t}} (C', \Sigma' \uplus \Sigma_F)) \wedge (((\sigma, \Sigma), (\sigma, \Sigma'), k) \models G * \mathsf{True}) \\ \quad \wedge\, (((\sigma, \Sigma'), (u', w'), C') \models q) \wedge (u', w') <_k (u, w) \qquad (<_k \text{ defined in Fig. 12}) \end{array}$$

The definition of $p \stackrel{G}{\Rrightarrow} q$ is similar to $p \Rrightarrow_k q$ defined in Fig. 13. In addition to executing the abstract code and decreasing the corresponding black tokens, $p \stackrel{G}{\Rrightarrow} q$ also requires the overall transition to satisfy G.

$$\frac{\begin{array}{c}\text{for all } f \in \mathit{dom}(\Pi): \quad \Pi(f) = (x, C) \quad \Gamma(f) = (y, \mathbb{C}) \quad \mathit{dom}(\Pi) = \mathit{dom}(\Gamma) \\ \mathcal{D}, R, G, I \vdash \{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y) \wedge \mathsf{arem}(\mathbb{C}) \wedge \blacklozenge(E_k, \ldots, E_1)\}C\{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge \mathsf{arem}(\textbf{skip})\} \\ \forall \mathsf{t}, \mathsf{t}'.\ \mathsf{t} \neq \mathsf{t}' \Longrightarrow G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'} \quad \mathsf{wffAct}(R, \mathcal{D}) \quad P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D}) \quad P \vee \mathsf{Enabled}(\mathcal{D}) \Rightarrow I\end{array}}{\mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma} \ (\textsc{obj})$$

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

$$\frac{\vdash [p]C[q'] \quad q' \Rrightarrow_k q \quad (\llbracket p \rrbracket \ltimes_k \llbracket q \rrbracket) \Rightarrow G * \mathsf{True} \quad I \triangleright G \quad p \vee q \Rightarrow I * \mathsf{true}}{\mathcal{D}, [I], G, I \vdash \{p\}\langle C \rangle\{q\}} \ (\textsc{atom})$$

$$\frac{\mathcal{D}, [I], G, I \vdash \{p\}\langle C \rangle\{q\} \quad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \quad I \triangleright R}{\mathcal{D}, R, G, I \vdash \{p\}\langle C \rangle\{q\}} \ (\textsc{atom-r}) \qquad \frac{\mathcal{D}, R, G, I \vdash \{p\}C\{q\}}{\mathcal{D}, R, G, I \vdash \{\lfloor p \rfloor_{\Diamond}\}C\{\lfloor q \rfloor_{\Diamond}\}} \ (\textsc{hide-}\Diamond)$$

$$\frac{p \Rightarrow (E = \mathbb{E}) * I \quad \mathsf{Sta}(p, R * \mathsf{Id}) \quad I \triangleright \{R, G\}}{\mathcal{D}, R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\textbf{return } \mathbb{E})\}\textbf{return } E\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\textbf{skip})\}} \ (\textsc{ret})$$

$$\frac{\vdash [p]C[q] \quad \mathsf{Sta}(r, R * \mathsf{Id}) \quad I \triangleright \{G, R\} \quad r \Rightarrow I * \mathsf{true}}{\mathcal{D}, R, G, I \vdash \{p * r\}C\{q * r\}} \ (\textsc{prim}) \qquad \frac{\mathcal{D}, R, G, I \vdash \{p\}C_1\{r\} \quad \mathcal{D}, R, G, I \vdash \{r\}C_2\{q\}}{\mathcal{D}, R, G, I \vdash \{p\}C_1; C_2\{q\}} \ (\textsc{seq})$$

$$\frac{p \Rightarrow (B = B) * I \quad \mathcal{D}, R, G, I \vdash \{p \wedge B\}C_1\{q\} \quad \mathcal{D}, R, G, I \vdash \{p \wedge \neg B\}C_2\{q\}}{\mathcal{D}, R, G, I \vdash \{p\}\textbf{if } (B)\ C_1 \textbf{ else } C_2\{q\}} \ (\textsc{if})$$

$$\frac{\mathcal{D}, R, G, I \vdash \{p\}C\{q\}}{\mathcal{D}, R, G, I \vdash \{p * r\}C\{q * r\}} \ (\textsc{frm}) \qquad \frac{\mathcal{D}, R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_1)\}C\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_2)\}}{\mathcal{D}, R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_1; \mathbb{C}_3)\}C\{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_2; \mathbb{C}_3)\}} \ (\textsc{arem})$$

$$\frac{\begin{array}{c}p' \overset{G}{\Rrightarrow} p \quad R' \Rightarrow R \quad \mathcal{D}, R, G, I \vdash \{p\}C\{q\} \quad q \overset{G}{\Rrightarrow} q' \quad G \Rightarrow G' \\ p' \vee q' \Rightarrow I' * \mathsf{true} \quad I' \triangleright \{G', R'\} \quad \mathsf{Sta}(\{p', q'\}, R * \mathsf{Id}) \quad \mathsf{Enabled}(\mathcal{D}) \Rightarrow I \quad \mathsf{wffAct}(R, \mathcal{D})\end{array}}{\mathcal{D}, R', G', I' \vdash \{p'\}C\{q'\}} \ (\textsc{csq})$$

$$\frac{\begin{array}{c}\mathcal{D}, R, G, I \vdash \{p\}C\{q\} \\ x \notin \mathit{fv}(\mathcal{D}, R, G, I)\end{array}}{\mathcal{D}, R, G, I \vdash \{\exists x.\ p\}C\{\exists x.\ q\}} \ (\textsc{ex}) \qquad \frac{\begin{array}{c}\mathcal{D}, R, G, I \vdash \{p_1\}C\{q_1\} \\ \mathcal{D}, R, G, I \vdash \{p_2\}C\{q_2\}\end{array}}{\mathcal{D}, R, G, I \vdash \{p_1 \wedge p_2\}C\{q_1 \wedge q_2\}} \ (\textsc{conj}) \qquad \frac{\begin{array}{c}\mathcal{D}, R, G, I \vdash \{p_1\}C\{q_1\} \\ \mathcal{D}, R, G, I \vdash \{p_2\}C\{q_2\}\end{array}}{\mathcal{D}, R, G, I \vdash \{p_1 \vee p_2\}C\{q_1 \vee q_2\}} \ (\textsc{disj})$$

Figure 21. LRG-style inference rules.

B. Logic Soundness Proofs

In this appendix, we give the detailed proofs of Theorem 2 following the unified proof framework in Fig. 18.

- Appendix B.1 defines the judgment semantics $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$.
- Appendix B.2 shows the proofs of ⑨ of Fig. 18, i.e., the logic rules are sound with respect to the judgment semantics.
- Appendix B.3 defines the "common simulation" $\Pi \precsim_{P'} \Pi'$. Appendix B.3.1 shows the proofs of ⑦ and ⑧ of Fig. 18, i.e., the judgment semantics $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ implies instantiations of the common simulation.
- Appendix B.4 shows the core proofs for the gray box at the center of Fig. 18, i.e., the common simulation implies $\Pi \sqsubseteq_{P'} \Pi'$, the contextual refinement under fair scheduling.
- Appendix B.5 shows the proofs of ① and ② of Fig. 18, i.e., deadlock-freedom/starvation-freedom and linearizability can be characterized by the contextual refinements.

B.1 Judgment Semantics

The judgment semantics $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ is based on a simulation between Π and Γ, with four well-founded metrics: M, ξ, $\mathbb{M}$ and u. The metric M is to ensure that the current thread t must fulfill its definite action $\mathcal{D}_\mathsf{t}$ in a finite number of steps. If thread t is blocked, the metric ξ specifies the set of the environment threads that t is waiting for. It shrinks when an environment thread t' finishes definite action $\mathcal{D}_{\mathsf{t}'}$. The metric $\mathbb{M}$ corresponds to the number of white tokens $\lozenge$, which is to ensure that thread t progresses on its own when ξ becomes empty. The last metric u corresponds to the tuple of black tokens. It bounds the number of actions made by thread t which could delay the progress of its environment threads.

Definition 5. $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$ iff, for any $f \in \mathit{dom}(\Pi)$, for any σ and Σ, for any t, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, \mathbb{C})$ and $(\sigma, \Sigma) \models P_\mathsf{t} * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$, there exist three well-founded metrics u, $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID})$ such that

$$\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_\xi (P * \mathsf{own}(x) * \mathsf{own}(y)).$$

Here $\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_\xi Q$ is co-inductively defined as follows.
Whenever $\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_\xi Q$ holds, then the following hold:

(1) Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathsf{t} \notin \xi$.
For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
(2) If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \perp \Sigma_F$, there exist $\mathbb{E}$ and Σ' such that
 (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_\mathsf{t}^* (\mathbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
 (b) $(\sigma, \Sigma') \models Q_\mathsf{t}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
 (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_\mathsf{t} * \mathsf{True}$.
(3) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_\mathsf{t} \mathbf{abort}$.
(4) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_\mathsf{t} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{M}'$, M' and ξ' such that
 (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
 (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_\mathsf{t}^* (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
 (c) $\mathcal{D}, R, G \models_\mathsf{t} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
 (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_\mathsf{t} * \mathsf{True}$, and
 (e) either $u' <_k u$,
 or $u' = u$ and $k = 0$ and $\mathbb{M}' < \mathbb{M}$,
 or $u' = u$ and $k = 0$ and $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
 (f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle [\mathcal{D}_\mathsf{t}] \rangle * \mathsf{True}$, then $M' < M$.
(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_\mathsf{t} * \mathsf{Id}$, then there exist u', $\mathbb{M}'$, M', ξ_d and ξ' such that
 (a) $\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
 (b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and
 (c) if $k = 0$, then either $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
 (d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_\mathsf{t}) * \mathsf{true}$, then $M' \leq M$.

Definition 6. $\mathcal{D}, R, G \models \{p\}C\{q\}$ iff, for any σ, Σ, u, w and $\mathbb{C}$, for any t, if $((\sigma, \Sigma), (u, w), \mathbb{C}) \models p_\mathsf{t}$, then

$$\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, ((0,0), |C|), (0, |C|), w, \mathsf{height}(C)) \Downarrow_\emptyset q.$$

Here $\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathtt{ws}, ws, w, \mathcal{H}) \Downarrow_\xi q$ is co-inductively defined as follows.
Whenever $\mathcal{D}, R, G \models_\mathsf{t} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathtt{ws}, ws, w, \mathcal{H}) \Downarrow_\xi q$ holds, then the following hold:

(1) Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathsf{t} \notin \xi$.
For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
If $\xi \neq \emptyset$, then $|\mathtt{ws}| > 1$.
(2) If $C = \mathbf{skip}$, then for any Σ_F such that $\Sigma \perp \Sigma_F$, there exist $\mathbb{C}'$ and Σ' such that
 (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_\mathsf{t}^* (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
 (b) $((\sigma, \Sigma'), (u, w), \mathbb{C}') \models q_\mathsf{t}$, and
 (c) $\mathtt{ws} = ((0,0), 0)$ and $ws = (0,0)$ and $\xi = \emptyset$, and
 (d) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_\mathsf{t} * \mathsf{True}$.

(3) If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \bot \Sigma_F$, there exist $\mathbb{E}$ and Σ' such that
   (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
   (b) $((\sigma, \Sigma'), (u, w), \mathbf{skip}) \models q_{\mathsf{t}}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
   (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True}$.

(4) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$.

(5) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$, then there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{ws}'$, ws', w' and ξ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
   (c) $\mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, and
   (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, and
   (e) either $u' <_k u$,
       or $u' = u$ and $k = 0$ and $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$,
       or $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
   (f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle [\mathcal{D}_{\mathsf{t}}] \rangle * \mathsf{True}$, then $ws' <_{\mathcal{H}} ws$.

(6) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist u', $\mathbb{ws}'$, ws', w', ξ_d and ξ' such that
   (a) $\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, and
   (b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and
   (c) if $k = 0$, then either $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$, or $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi_d = \emptyset$; and
   (d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $ws' \leq_{\mathcal{H}} ws$.

Below we also define the semantics for the sequential judgment used in the ATOM rule. Note that C only accesses the concrete memory σ, therefore we require the other components in the full state (i.e., u, w, $\mathbb{C}$ and Σ) should remain unchanged during the execution of C.

Definition 7 (SL judgment semantics, total correctness)**.** $\models [p]C[q]$ iff, for all σ, Σ, u, w and $\mathbb{C}$, for any t, if $((\sigma, \Sigma), (u, w), \mathbb{C}) \models p_{\mathsf{t}}$, the following are true:

1. for any σ', if $(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\mathbf{skip}, \sigma')$, then $((\sigma', \Sigma), (u, w), \mathbb{C}) \models q_{\mathsf{t}}$;
2. $(C, \sigma) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$;
3. $(C, \sigma) \not\longrightarrow_{\mathsf{t}}^{\omega} \cdot$.

Lemma 8. If $\vdash [p]C[q]$, then $\models [p]C[q]$.

Definition 9 (Locality)**.**
$\mathsf{Locality}(C)$ iff, for any σ_1 and σ_2, let $\sigma = \sigma_1 \uplus \sigma_2$, then the following hold:

1. (Safety monotonicity) If $(C, \sigma_1) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$, then $(C, \sigma) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$.
2. (Termination monotonicity) If $(C, \sigma_1) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$ and $(C, \sigma_1) \not\longrightarrow_{\mathsf{t}}^{\omega} \cdot$, then $(C, \sigma) \not\longrightarrow_{\mathsf{t}}^{\omega} \cdot$.
3. (Frame property) For any n and σ', if $(C, \sigma_1) \not\longrightarrow_{\mathsf{t}}^{*} \mathbf{abort}$ and $(C, \sigma) \longrightarrow_{\mathsf{t}}^{n} (C', \sigma')$, then there exists σ'_1 such that $\sigma' = \sigma'_1 \uplus \sigma_2$ and $(C, \sigma_1) \longrightarrow_{\mathsf{t}}^{n} (C', \sigma'_1)$.

B.1.1 Instantiating Metrics and Well-Founded Orders

The judgment semantics in Definition 6 can be viewed as an instantiation of the simulation in Definition 5. The key is to instantiate the metrics $\mathbb{M}$ and M in $\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$. Below we take the instantiation for M as an example.

$$\begin{array}{rrcl} (\mathit{Metric}) & M & ::= & (ws, \mathcal{H}) \\ (\mathit{WfStack}) & ws & ::= & (w, n) \mid (w, n) :: ws \\ (\mathit{StkHeight}) & \mathcal{H} & \in & \mathit{Nat} \end{array}$$

For each single thread, its metric ws is usually a list of (w, n) pairs, where w is the while-specific metric (which is left to be instantiated later) and n is a natural number specifying the "code size" as in Liang et al.'s work [23]. We let the threaded metric ws be a list (a stack actually) to allow different while-specific metrics for nested loops. That is, when entering a loop, we can push a (w, n) pair to the ws stack; and when exiting the loop, we pop the pair out of ws.

The threaded metric ws follows the dictionary order. However, the usual dictionary order over lists is not well-founded (consider $B > AB > AAB > AAAB > \ldots$ in a dictionary). To address this issue, we introduce a bound of the list length (stack height), $\mathcal{H}$, and define the well-founded order $\prec_{\mathcal{H}}$ by requiring the length of the lists should be not larger than $\mathcal{H}$. Intuitively, the stack height $\mathcal{H}$ represents the maximal depth of nested loops, so it can be determined for any given program. The well-founded orders $M' < M$ and $ws' \prec_{\mathcal{H}} ws$ are defined as follows.

$$\frac{ws' \prec_{\mathcal{H}} ws \qquad \mathcal{H}' = \mathcal{H}}{(ws', \mathcal{H}') < (ws, \mathcal{H})}$$

$$\begin{array}{rcl} ws' \prec_{\mathcal{H}} ws & \text{iff} & (ws' \lessdot ws) \wedge (|ws'| \leq \mathcal{H}) \wedge (|ws| \leq \mathcal{H}) \\ ws' \preccurlyeq_{\mathcal{H}} ws & \text{iff} & (ws' \prec_{\mathcal{H}} ws) \vee (ws' = ws) \end{array}$$

$$\frac{(w',n') < (w,n)}{(w',n') \prec\!\!\!\prec (w,n)} \qquad \frac{(w',n') < (w,n)}{(w',n') :: ws_1' \prec\!\!\!\prec (w,n) :: ws_1}$$

$$\frac{(w',n') = (w,n) \qquad ws_1' \prec\!\!\!\prec ws_1}{(w',n') :: ws_1' \prec\!\!\!\prec (w,n) :: ws_1}$$

$$\frac{(w',n') < (w,n)}{(w',n') :: ws_1' \prec\!\!\!\prec (w,n)} \qquad \frac{(w',n') \le (w,n)}{(w',n') \prec\!\!\!\prec (w,n) :: ws_1}$$

Here $|ws|$ is the length of ws, which is defined as follows:

$$\begin{array}{rcl} |(w,n)| & = & 1 \\ |(w,n) :: ws| & = & 1 + |ws| \end{array}$$

The well-founded order over the (w,n) pairs is a usual dictionary order below, where the order over w is instantiated later depending on the type of w.

$$\begin{array}{lcl} (w',n') < (w,n) & \text{iff} & (w' < w) \vee (w' = w \wedge n' < n) \\ (w',n') = (w,n) & \text{iff} & (w' = w) \wedge (n' = n) \\ (w',n') \le (w,n) & \text{iff} & (w',n') < (w,n) \vee (w',n') = (w,n) \end{array}$$

Lemma 10 (Well-foundedness)**.** The relations $M' < M$ and $ws' <_{\mathcal{H}} ws$ defined above are both well-founded relations.

The judgment semantics in Definition 6 has more restrictions on the well-founded order over ws. We define the order $ws' <_{\mathcal{H}} ws$ below, which is stronger than the more general order $ws' \prec_{\mathcal{H}} ws$ above.

$$\begin{array}{lcl} ws' <_{\mathcal{H}} ws & \text{iff} & (ws' \ll ws) \wedge (|ws'| \le \mathcal{H}) \wedge (|ws| \le \mathcal{H}) \\ ws' \le_{\mathcal{H}} ws & \text{iff} & (ws' <_{\mathcal{H}} ws) \vee (ws' = ws) \end{array}$$

$$\frac{(w',n') < (w,n)}{(w',n') \ll (w,n)} \qquad \frac{(w',n') = (w,n) \qquad ws_1' \ll ws_1}{(w',n') :: ws_1' \ll (w,n) :: ws_1}$$

$$\frac{(w',n') < (w,n)}{(w',n') :: ws_1' \ll (w,n)} \qquad \frac{(w',n') = (w,n)}{(w',n') \ll (w,n) :: ws_1}$$

$$\frac{(w_0',n_0') = (w_0,n_0) \qquad (w',n') < (w,n)}{(w_0',n_0') :: (w',n') :: ws_1' \ll (w_0,n_0) :: (w,n) :: ws_1}$$

Height $\mathcal{H}$. As we said, the stack height $\mathcal{H}$ represents the maximal depth of nested loops. For any given program C, we can determine the stack height using a function height defined below.

$$\begin{array}{rcl} \mathsf{height}(\textbf{skip}) & \stackrel{\text{def}}{=} & 1 \\ \mathsf{height}(\textbf{return}\ E) & \stackrel{\text{def}}{=} & 1 \\ \mathsf{height}(c) & \stackrel{\text{def}}{=} & 1 \\ \mathsf{height}(\langle C \rangle) & \stackrel{\text{def}}{=} & 1 \\ \mathsf{height}(C_1; C_2) & \stackrel{\text{def}}{=} & \mathit{max}\{\mathsf{height}(C_1), \mathsf{height}(C_2)\} \\ \mathsf{height}(\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2) & \stackrel{\text{def}}{=} & \mathit{max}\{\mathsf{height}(C_1), \mathsf{height}(C_2)\} \\ \mathsf{height}(\textbf{while}\ (B)\{C\}) & \stackrel{\text{def}}{=} & \mathsf{height}(C) + 1 \end{array}$$

Initial code size. In Definition 6, the judgment semantics initially takes the static code size $|C|$ defined below as the second dimension of ws and $\mathtt{ws}$.

$$\begin{array}{rcl} |\textbf{skip}| & \stackrel{\text{def}}{=} & 0 \\ |\textbf{return}\ E| & \stackrel{\text{def}}{=} & 1 \\ |c| & \stackrel{\text{def}}{=} & 1 \\ |\langle C \rangle| & \stackrel{\text{def}}{=} & 1 \\ |C_1; C_2| & \stackrel{\text{def}}{=} & |C_1| + |C_2| + 1 \\ |\textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2| & \stackrel{\text{def}}{=} & \mathit{max}\{|C_1|, |C_2|\} + 1 \\ |\textbf{while}\ (B)\{C\}| & \stackrel{\text{def}}{=} & 1 \end{array}$$

Example of ws. Below we use a simple example to show how we assign a proper ws to each state during an execution. In the code below, we assign different labels to different layers of a nested while loop. The initial code is `while(i > 0) i--;`. The loop is labeled with 1 and its body code is labeled with 2. After the loop is unfolded, we use the syntax while to be distinguished from the original `while` which has not been unfolded.

In the ws below, the first dimension specifies the number of iterations left to unfold, and the second dimension specifies the "code size" at each layer.

		C	σ	ws
1		$\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 2$	$(0,1)$
2	$\rightarrow$	$\texttt{i--}^{2}$; $\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 2$	$(0,0) :: (1,2)$
3	$\rightarrow$	$\texttt{skip}^{2}$; $\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 1$	$(0,0) :: (1,1)$
4	$\rightarrow$	$\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 1$	$(0,0) :: (1,0)$
5	$\rightarrow$	$\texttt{i--}^{2}$; $\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 1$	$(0,0) :: (0,2)$
6	$\rightarrow$	$\texttt{skip}^{2}$; $\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 0$	$(0,0) :: (0,1)$
7	$\rightarrow$	$\texttt{while}^{1}\texttt{(i > 0) i--}^{2}$;	$\texttt{i} = 0$	$(0,0) :: (0,0)$
8	$\rightarrow$	$\texttt{skip}^{1}$;	$\texttt{i} = 0$	$(0,0)$

Initially, ws is $(0,1)$: the first dimension is 0 because we have not started to unfold the loop, and the second dimension is 1 because the code size of the whole loop is 1. After one step of the loop, ws becomes $(0,0) :: (1,2)$. Since we have unfolded the loop, the ws stack contains two pairs now. In the second pair, the first dimension is 1 because the loop needs only one more iteration to finish (i.e., we only need to unfold it one more time). Its second dimension is 2 because the size of the loop body code is 2. After the next step, this dimension decreases. At the step of line 5, we unfold the loop again. So the first dimension of the second pair decreases to 0, saying that we do not need to unfold the loop anymore. Finally, at the step of line 8, the loop finishes, thus we pop out the second pair of the ws stack.

B.2 Soundness of the Inference Rules

In this section, we prove Lemma 11 by induction over the derivation.

Lemma 11 (⑨ in Fig. 18)**.** If $\mathcal{D}, R, G, I \vdash \{P\}\Pi : \Gamma$, then $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$.

Proof. By induction over derivation. By Lemma 12, we only need to prove the following:

> If $dom(\Pi) = dom(\Gamma)$ and for all $f \in dom(\Pi)$ such that $\Pi(f) = (x, C)$ and $\Gamma(f) = (y, \mathbb{C})$, we have
> $\mathcal{D}, R, G \models \{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y) \wedge \mathsf{arem}(\mathbb{C}) \wedge \blacklozenge(E_k, \ldots, E_1)\}\ C\ \{P * \mathsf{own}(x) * \mathsf{own}(y) \wedge \mathsf{arem}(\textbf{skip})\}$,
> then $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$.
>
> (B.1)

By co-induction. □

Lemma 12. If

1. $\mathcal{D}, R, G, I \vdash \{p\}C\{q\}$;
2. $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$;
3. for any t and t' such that $\mathsf{t} \neq \mathsf{t}'$, we have $G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$;

then $\mathcal{D}, R, G \models \{p\}C\{q\}$.

In the following subsections, we prove Lemma 12 by induction over the derivation.

B.2.1 The WHL Rule

Lemma 13 (WHL-Sound)**.** If

1. $p \wedge B \Rightarrow p'$; $p \wedge B \wedge Q * \mathsf{true} \Rightarrow p' * (\Diamond \wedge \mathsf{emp})$;
2. $\mathcal{D}, R, G \models \{p'\}C\{p\}$;
3. $p \Rightarrow (B = B) * J$; $J \Rightarrow I$; $\mathsf{Sta}(J, R \vee G)$; $Q \Rightarrow I$;
4. $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$; $\mathcal{D}' \leqslant \mathcal{D}$; $\mathsf{wffAct}(R, \mathcal{D}')$;
5. $I \triangleright \{R, G\}$; $\mathsf{Sta}(p, R * \mathsf{Id})$; $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$;
6. for any t and t' such that $\mathsf{t} \neq \mathsf{t}'$, we have $G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$;

then $\mathcal{D}, R, G \models \{p\}\textbf{while}\ (B)\{C\}\{p \wedge \neg B\}$.

Proof. Let $\mathcal{H} = \mathsf{height}(\textbf{while}\ (B)\{C\}) = \mathsf{height}(C) + 1$. We know $|\textbf{while}\ (B)\{C\}| = 1$. Let

$$\mathbb{ws} = ((0,0),1) \text{ and } ws = (0,1) \text{ and } \xi = \emptyset.$$

Below we prove: for any t, for any σ, Σ, u, w and $\mathbb{C}$,
if $((\sigma, \Sigma), (u, w), \mathbb{C}) \models p_{\mathsf{t}}$, then

$$\mathcal{D}, R, G \models_{\mathsf{t}} (\textbf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi}\ p \wedge \neg B\ .$$

By co-induction. Suppose $\sigma = (s, h)$. Since $p \Rightarrow (B = B) * I$, we know

$$(\sigma, \Sigma) \models I * \mathsf{true}, \text{ and either } [\![B]\!]_s = \textbf{true} \text{ or } [\![B]\!]_s = \textbf{false}\ .$$

Since $p \Rightarrow J * \mathsf{true}$, we know

$$(\sigma, \Sigma) \models J * \mathsf{true}\ .$$

We only need to prove the following (1)(2)(3)(4)(5)(6).

(1) For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
Proof: It is vacantly true.

(2) If $\textbf{while}\ (B)\{C\} = \textbf{skip}$, then ....
Proof: It is vacantly true.

(3) If $\textbf{while}\ (B)\{C\} = \mathbf{E}[\,\textbf{return}\ E\,]$, then ....
Proof: It is vacantly true.

(4) For any σ_F, $(\textbf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \textbf{abort}$.
Proof: It holds because $[\![B]\!]_s$ is not undefined.

(5) If $(\textbf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$, then there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{ws}'$, ws', w' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
  (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
  (c) $\mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'}\ p \wedge \neg B$, and
  (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, and
  (e) either $u' <_k u$,
    or $u' = u$ and $k = 0$ and $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$,
    or $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and

(f) if $((\sigma,\Sigma),(\sigma',\Sigma')) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $ws' <_{\mathcal{H}} ws$.
Proof: We have two cases depending on whether $[\![B]\!]_s$ is **true** or **false**.
(I) If $[\![B]\!]_s =$ **true**, we know $\sigma'' = \sigma \uplus \sigma_F$ and $(\textbf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$.
Also we know $((\sigma,\Sigma),(u,w),\mathbb{C}) \models p_{\mathsf{t}} \wedge B$. From $p \wedge B \Rightarrow p'$ and $\mathcal{D},R,G \models \{p'\}C\{p\}$, let $\mathbb{ws}_1 = ((0,0),|C|)$ and $ws_1 = (0,|C|)$, we get:

$$\mathcal{D},R,G \models_{\mathsf{t}} (C,\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w, \mathsf{height}(C)) \Downarrow_{\emptyset} p\ .$$

Let

$$ws' = (0,0) :: (w, |C|+1)\ ,$$

we know $ws' <_{\mathcal{H}} ws$. Let

$$k_s = f(\sigma,\Sigma) \ \text{ and } \ \xi_0 = \{\mathsf{t}'' \mid (\mathsf{t}'' \neq \mathsf{t}) \wedge ((\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}''}) * \mathsf{true})\}\ .$$

Since $J \Rightarrow (R,G\colon \mathcal{D}' \xrightarrow{f} Q)$, we have two cases:

- $(\sigma,\Sigma) \models \exists \mathsf{t}' \neq \mathsf{t}.\ \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true}$.
  Then we know $\xi_0 \neq \emptyset$. By Lemma 14, let
  $$\mathbb{ws}' = ((0,0),0) :: ((k_s,w),0)\ .$$
  we know
  $$\mathcal{D},R,G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}) \Downarrow_{\xi_0} p \wedge \neg B\ .$$
  Also we know $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$.
- $(\sigma,\Sigma) \models Q_{\mathsf{t}} * \mathsf{true}$.
  By Lemma 14, let
  $$\mathbb{ws}' = ((0,0),0) :: ((k_s,w),|C|+1)\ ,$$
  we know
  $$\mathcal{D},R,G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}) \Downarrow_{\xi_0} p \wedge \neg B\ .$$
  Also, $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ holds.

(II) If $[\![B]\!]_s =$ **false**, we know $(\textbf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (\textbf{skip}, \sigma \uplus \sigma_F)$.
By (SKIP) rule, let

$$\mathbb{ws}' = ((0,0),0) \text{ and } ws' = (0,0)\ ,$$

we know

$$\mathcal{D},R,G \models_{\mathsf{t}} (\textbf{skip},\sigma) \preceq (\mathbb{C},\Sigma) \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}) \Downarrow_{\emptyset} p \wedge \neg B\ .$$

Also we know $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $ws' <_{\mathcal{H}} ws$.
Since $(\sigma,\Sigma,i) \models I * \mathsf{true}$, we know

$$((\sigma,\Sigma),(\sigma,\Sigma),0) \models [I] * \mathsf{True}\ .$$

Since $I \triangleright G$, we know

$$((\sigma,\Sigma),(\sigma,\Sigma),0) \models G_{\mathsf{t}} * \mathsf{True}\ .$$

(6) If $((\sigma,\Sigma),(\sigma',\Sigma'),k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist u', $\mathbb{ws}'$, ws', w', ξ_d and ξ' such that
(a) $\mathcal{D},R,G \models_{\mathsf{t}} (\textbf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C},\Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} p \wedge \neg B$, and
(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma,\Sigma),(\sigma',\Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and
(c) if $k = 0$, then either $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$, or $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi_d = \emptyset$; and
(d) if $k = 0$ and $(\sigma,\Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$,
then $ws' \leq_{\mathcal{H}} ws$.
Proof: Since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know there exist u' and w' such that

$$((\sigma',\Sigma'),(u',w'),\mathbb{C}) \models p_{\mathsf{t}} \ \text{ and } \ u' \approx_k u \ \text{ and } \ k = 0 \Longrightarrow w' = w\ .$$

By the co-induction hypothesis, we get

$$\mathcal{D},R,G \models_{\mathsf{t}} (\textbf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C},\Sigma') \diamond (u, \mathbb{ws}, ws, w', \mathcal{H}) \Downarrow_{\xi} p \wedge \neg B\ .$$

If $k = 0$, we know $w' = w$ and $\xi_d = \emptyset$.

Thus we are done. □

We define:

$$\mathsf{head}(\mathbb{ws}) \stackrel{\text{def}}{=} \begin{cases} ((n_1,n_2),n_3) & \text{if } \mathbb{ws} = ((n_1,n_2),n_3) \\ ((n_1,n_2),n_3) & \text{if } \mathbb{ws} = ((n_1,n_2),n_3) :: \mathbb{ws}' \end{cases}$$

$$\mathsf{inchead}(\mathbb{ws}, ((k_1,k_2),k_3)) \stackrel{\text{def}}{=} \begin{cases} ((n_1+k_1, n_2+k_2), n_3+k_3) & \text{if } \mathbb{ws} = ((n_1,n_2),n_3) \\ ((n_1+k_1, n_2+k_2), n_3+k_3) :: \mathbb{ws}' & \text{if } \mathbb{ws} = ((n_1,n_2),n_3) :: \mathbb{ws}' \end{cases}$$

Lemma 14. If

1. $p \wedge B \Rightarrow p'$; $p \wedge B \wedge Q * \mathsf{true} \Rightarrow p' * (\Diamond \wedge \mathsf{emp})$;
2. $\mathcal{D},R,G \models \{p'\}C\{p\}$;
3. $p \Rightarrow (B = B) * J$; $J \Rightarrow I$; $\mathsf{Sta}(J, R \vee G)$; $Q \Rightarrow I$;

4. $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$; $\mathcal{D}' \leqslant \mathcal{D}$; $\mathsf{wffAct}(R, \mathcal{D}')$;
5. $I \triangleright \{R, G\}$; $\mathsf{Sta}(p, R * \mathsf{Id})$; $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$;
6. for any t and t' such that $\mathsf{t} \neq \mathsf{t}'$, we have $G_\mathsf{t} \Rightarrow R_{\mathsf{t}'}$;
7. $\mathcal{D}, R, G \models_\mathsf{t} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$;
8. $(\sigma, \Sigma) \models J * \mathsf{true}$; $\mathsf{height}(C) = \mathcal{H}$; $\mathsf{head}(\mathbb{ws}_1) = ((n_1, n_2), n_3)$; $f(\sigma, \Sigma, i) = k_s$; $w_1 \leq w$;
9. $\xi_0 = \{\mathsf{t}' \mid (\mathsf{t}' \neq \mathsf{t}) \wedge ((\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true})\}$; $\xi = \xi_0 \cup \xi_1$;
10. $ws = (0, 0) :: \mathsf{inchead}(ws_1, (w_1, 1))$;
11. one of the following holds:
    - $\xi_0 \neq \emptyset$; $\mathbb{ws} = ((0, 0), 0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), -n_3))$;
    - $(\sigma, \Sigma) \models Q_\mathsf{t} * \mathsf{true}$; $\mathbb{ws} = ((0, 0), 0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), 1))$;

then $\mathcal{D}, R, G \models_\mathsf{t} (C_1; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H} + 1) \Downarrow_\xi p \wedge \neg B$.

Proof. By co-induction. We only need to prove the following (1)(2)(3)(4)(5).

(1) For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
Proof: From

$$\mathcal{D}, R, G \models_\mathsf{t} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p\ ,$$

we know for any $\mathsf{t}' \in \xi_1$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$. Since $\mathcal{D}' \leqslant \mathcal{D}$, we know for any $\mathsf{t}' \in \xi_0$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$. Thus we are done.

(2) If $(C_1; \mathsf{while}\ (B)\{C\}) = \mathbf{skip}$, then ...
Proof: It is vacantly true.

(3) For any σ_F, $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \not\longrightarrow_\mathsf{t} \mathbf{abort}$.
Proof: From

$$\mathcal{D}, R, G \models_\mathsf{t} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p\ ,$$

we know $(C_1, \sigma \uplus \sigma_F) \not\longrightarrow_\mathsf{t} \mathbf{abort}$. By the operational semantics, we are done.

(4) If $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_\mathsf{t} (C', \sigma'')$, then there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{ws}'$, ws', w' and ξ' such that
(a) $\sigma'' = \sigma' \uplus \sigma_F$, and
(b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_\mathsf{t}^* (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
(c) $\mathcal{D}, R, G \models_\mathsf{t} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', w', \mathcal{H} + 1) \Downarrow_{\xi'} p \wedge \neg B$, and
(d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_\mathsf{t} * \mathsf{True}$, and
(e) either $u' <_k u$,
or $u' = u$ and $k = 0$ and $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$ and $w' = w$,
or $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_\mathsf{t}]\rangle * \mathsf{True}$, then $ws' <_{\mathcal{H}+1} ws$.
Proof: We have two cases depending on whether C_1 is **skip** or not.
(I) If $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_\mathsf{t} (C_1'; \mathsf{while}\ (B)\{C\}, \sigma'')$, then $(C_1, \sigma \uplus \sigma_F) \longrightarrow_\mathsf{t} (C_1', \sigma'')$. From $\mathcal{D}, R, G \models_\mathsf{t} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, we know there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{ws}_1'$, ws_1', w_1' and ξ_1' such that
(A) $\sigma'' = \sigma' \uplus \sigma_F$, and
(B) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_\mathsf{t}^* (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
(C) $\mathcal{D}, R, G \models_\mathsf{t} (C_1', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}_1', ws_1', w_1', \mathcal{H}) \Downarrow_{\xi_1'} p$, and
(D) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_\mathsf{t} * \mathsf{True}$, and
(E) either $u' <_k u$,
or $u' = u$ and $k = 0$ and $\mathbb{ws}_1' <_\mathcal{H} \mathbb{ws}_1$ and $w_1' = w_1$,
or $u' = u$ and $k = 0$ and $\mathbb{ws}_1' = \mathbb{ws}_1$ and $w_1' = w_1$ and $\xi_1 \neq \emptyset$ and $\xi_1 \subseteq \xi'$; and
(F) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_\mathsf{t}]\rangle * \mathsf{True}$, then $ws_1' <_\mathcal{H} ws_1$.
Since $((\sigma, \Sigma), (\sigma', \Sigma')) \models G_\mathsf{t} * \mathsf{True}$, $(\sigma, \Sigma) \models J * \mathsf{true}$, $J \Rightarrow I$, $I \triangleright G$ and $\mathsf{Sta}(J, G \vee R)$, we know

$$(\sigma', \Sigma') \models J * \mathsf{true}\ .$$

Suppose $k_s' = f(\sigma', \Sigma')$. Since $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$, we can prove

$$k = 0 \implies k_s' \leq k_s\ .$$

Also we have

$$(\sigma', \Sigma') \models Q_\mathsf{t} * \mathsf{true} \vee (\exists \mathsf{t}' \neq \mathsf{t}.\ \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true})\ .$$

Let

$$\xi_0' = \{\mathsf{t}' \mid (\mathsf{t}' \neq \mathsf{t}) \wedge ((\sigma', \Sigma') \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true})\}\ .$$

Since for any t' such that $\mathsf{t}' \neq \mathsf{t}$ we have $G_\mathsf{t} \Rightarrow R_{\mathsf{t}'}$, and since $\mathsf{wffAct}(R, \mathcal{D}')$, $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, we can prove:

$$k = 0 \implies \xi_0 \subseteq \xi_0'\ .$$

Let

$$ws' = (0, 0) :: \mathsf{inchead}(ws_1', (w_1, 1))\ .$$

We know: if $ws'_1 <_{\mathcal{H}} ws_1$, then $ws' <_{\mathcal{H}+1} ws$. If $w'_1 = w_1$, let $w' = w$; otherwise let $w' = w'_1$. Thus we know $w'_1 \leq w'$.
Suppose $\mathsf{head}(\mathbb{ws}'_1) = ((n'_1, n'_2), n'_3)$.

- If $\xi'_0 \neq \emptyset$, let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}'_1, ((k'_s, w_1), -n'_3)) \text{ and } \xi' = \xi'_0 \cup \xi'_1 \,.$$
Then by the co-induction hypothesis, we know
$$\mathcal{D}, R, G \models_{\mathsf{t}} (C'_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
Also, if $\mathbb{ws}'_1 \leq_{\mathcal{H}} \mathbb{ws}_1$, then $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$. Thus, we know: either $u' <_k u$, or $u' = u$ and $k = 0$ and $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$ and $w' = w$.
For the case that $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$, we know
$$\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), -n_3)) \text{ and } \xi_0 \neq \emptyset \text{ and } \xi = \xi_0 \cup \xi_1,$$
thus $\xi \neq \emptyset$.
  - Suppose $\mathbb{ws}'_1 <_{\mathcal{H}} \mathbb{ws}_1$, since $\mathbb{ws}' = \mathbb{ws}$, we know $n'_3 < n_3$. Then, from the definition of $<_{\mathcal{H}}$, we know $\mathbb{ws}_1 = ((n_1, n_2), n_3)$, thus $\xi_1 = \emptyset$. Thus $\xi_1 \subseteq \xi'_1$.
  - Suppose $\mathbb{ws}'_1 = \mathbb{ws}_1$. Then we know $\xi_1 \subseteq \xi'_1$.

  Thus $\xi \subseteq \xi'$.
- If $\xi'_0 = \emptyset$, then we know $(\sigma', \Sigma') \models Q_{\mathsf{t}} * \mathsf{true}$. Since $\xi_0 \subseteq \xi'_0$, we know $\xi_0 = \emptyset$. Thus we know $(\sigma, \Sigma) \models Q_{\mathsf{t}} * \mathsf{true}$ and $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), 1))$. Also we have $\xi = \xi_0 \cup \xi_1 = \xi_1$. Let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}'_1, ((k'_s, w_1), 1)) \text{ and } \xi' = \xi'_0 \cup \xi'_1 = \xi'_1 \,.$$
By the co-induction hypothesis, we know
$$\mathcal{D}, R, G \models_{\mathsf{t}} (C'_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B \,.$$
Also, if $\mathbb{ws}'_1 \leq_{\mathcal{H}} \mathbb{ws}_1$, then $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$. Thus, we know: either $u' <_k u$, or $u' = u$ and $k = 0$ and $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$ and $w' = w$.
For the case that $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$, we know $\mathbb{ws}'_1 = \mathbb{ws}_1$, thus $\xi \neq \emptyset$. Since $\xi_1 \subseteq \xi'_1$, we know $\xi \subseteq \xi'$.

(II) If $C_1 = \mathbf{skip}$ and $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$,
from $\mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, we know there exist $\mathbb{C}'$ and Σ' such that
  (A) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
  (B) $((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}}$, and
  (C) $\mathbb{ws}_1 = ((0,0),0)$ and $ws_1 = (0,0)$ and $\xi_1 = \emptyset$, and
  (D) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True}$.

Since $(\sigma, \Sigma) \models J * \mathsf{true}$, $J \Rightarrow I$, $I \triangleright G$ and $\mathsf{Sta}(J, G \vee R)$, we know
$$(\sigma, \Sigma') \models J * \mathsf{true} \,.$$
Suppose $k'_s = f(\sigma, \Sigma')$. Since $J \Rightarrow (R, G \colon \mathcal{D}' \xrightarrow{f} Q)$, we can prove
$$k'_s \leq k_s \,.$$
Also we have
$$(\sigma, \Sigma') \models Q_{\mathsf{t}} * \mathsf{true} \vee (\exists \mathsf{t}' \neq \mathsf{t}.\ \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true}) \,.$$
Let
$$\xi'_0 = \{\mathsf{t}' \mid (\mathsf{t}' \neq \mathsf{t}) \wedge ((\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true})\} \,.$$
Since for any t' such that $\mathsf{t}' \neq \mathsf{t}$ we have $G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$, and since $\mathsf{wffAct}(R, \mathcal{D}')$, $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, we can prove:
$$\xi_0 \subseteq \xi'_0 \,.$$
Let
$$ws' = (0,0) :: \mathsf{inchead}(ws_1, (w_1, 0)) \,.$$
Thus $ws' \leq_{\mathcal{H}+1} ws$. Let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k'_s, w_1), 0)) \,.$$
Thus $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$. Also, if $\mathbb{ws}' = \mathbb{ws}$, then $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), -n_3))$ and $\xi_0 \neq \emptyset$ and $\xi = \xi_0 \cup \xi_1$, thus $\xi \neq \emptyset$ and $\xi \subseteq \xi'_0$.
Below we prove:
$$\mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}', ws', w, \mathcal{H}+1) \Downarrow_{\xi'_0} p \wedge \neg B \,. \tag{B.2}$$

Proof: By co-induction. Suppose $\sigma = (s, h)$. Since $p \Rightarrow (B = B) * I$, we know
$$(\sigma, \Sigma') \models I * \mathsf{true}, \text{ and either } [\![B]\!]_s = \mathbf{true} \text{ or } [\![B]\!]_s = \mathbf{false} \,.$$
We only need to prove the following (*1*)(*2*)(*3*)(*4*)(*5*).
(*1*) For any $\mathsf{t}' \in \xi'_0$, we have $(\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
  Proof: Immediate from $\mathcal{D}' \leqslant \mathcal{D}$.
(*2*) If $\mathsf{while}\ (B)\{C\} = \mathbf{skip}$, then ...
  Proof: It is vacantly true.
(*3*) For any σ_F, $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$.
  Proof: It holds because $[\![B]\!]_s$ is not undefined.
(*4*) If $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$, then there exist σ', $\mathbb{C}''$, Σ'', k, u'', $\mathbb{ws}''$, ws'', w'' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
  (b) $(\mathbb{C}', \Sigma' \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (\mathbb{C}'', \Sigma'' \uplus \Sigma_F)$, and
  (c) $\mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}'', \Sigma'') \diamond (u'', \mathbb{ws}'', ws'', w'', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B$, and
  (d) $((\sigma, \Sigma'), (\sigma', \Sigma''), k) \models G_{\mathsf{t}} * \mathsf{True}$, and

(e) either $u'' <_k u$,
or $u'' = u$ and $k = 0$ and $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$ and $w'' = w$,
or $u'' = u$ and $k = 0$ and $\mathbb{ws}'' = \mathbb{ws}'$ and $w'' = w$ and $\xi_0' \neq \emptyset$ and $\xi_0' \subseteq \xi'$; and

(f) if $((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $ws'' <_{\mathcal{H}+1} ws'$.

Proof: We have two cases depending on whether $[\![B]\!]_s$ is **true** or **false**.

- If $[\![B]\!]_s =$ **true**, we know $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$. Also we know $((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}} \wedge B$.
  - If $\xi_0' \neq \emptyset$, we have two cases below:
  - If $(\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, we know
  $$((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}} \wedge B \wedge \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true} .$$
  Since $p \wedge B \wedge \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true} \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp})$, we know there exists w_1' such that $w_1' < w_1$ and
  $$((\sigma, \Sigma'), (u, w_1'), \mathbb{C}') \models p_{\mathsf{t}}' .$$
  From $\mathcal{D}, R, G \models \{p'\}C\{p\}$ and $\mathsf{height}(C) = \mathcal{H}$, let $ws_1' = (0, |C|)$ and $\mathbb{ws}_1' = ((0,0), |C|)$, we get:
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}_1', ws_1', w_1', \mathcal{H}) \Downarrow_{\emptyset} p .$$
  Let
  $$\mathbb{ws}'' = ((0,0),0) :: ((k_s', w_1'), 0) \text{ and } ws'' = (0,0) :: (w_1', |C|+1) ,$$
  By the co-induction hypothesis, we know
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', ws'', w, \mathcal{H}+1) \Downarrow_{\xi_0'} p \wedge \neg B .$$
  Also we have $ws'' <_{\mathcal{H}+1} ws'$ and $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$. Since $(\sigma, \Sigma') \models I * \mathsf{true}$, we can prove
  $$((\sigma, \Sigma'), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True} .$$
  - Otherwise, from $p \wedge B \Rightarrow p'$, we know
  $$((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}}' .$$
  From $\mathcal{D}, R, G \models \{p'\}C\{p\}$ and $\mathsf{height}(C) = \mathcal{H}$, let $\mathbb{ws}_1' = ((0,0), |C|)$ and $ws_1' = (0, |C|)$, we get:
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}_1', ws_1', w_1, \mathcal{H}) \Downarrow_{\emptyset} p .$$
  Let
  $$\mathbb{ws}'' = ((0,0),0) :: ((k_s', w_1), 0) \text{ and } ws'' = (0,0) :: (w_1, |C|+1) .$$
  By the co-induction hypothesis, we know
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', ws'', w, \mathcal{H}+1) \Downarrow_{\xi_0'} p \wedge \neg B .$$
  Also we know $\mathbb{ws}'' = \mathbb{ws}'$ and $\xi_0' \neq \emptyset$. Since $(\sigma, \Sigma') \models I * \mathsf{true}$, we can prove:
  $$((\sigma, \Sigma'), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True} .$$
  - If $\xi_0' = \emptyset$, then we know $(\sigma, \Sigma') \models Q_{\mathsf{t}} * \mathsf{true}$. Thus we know
  $$((\sigma, \Sigma'), (u, w_1), \mathbb{C}') \models p_{\mathsf{t}} \wedge B \wedge Q_{\mathsf{t}} * \mathsf{true} .$$
  Since $p \wedge B \wedge Q * \mathsf{true} \Rightarrow p' * (\Diamond(1) \wedge \mathsf{emp})$, we know there exists w_1' such that $w_1' < w_1$ and
  $$((\sigma, \Sigma', i), (u, w_1'), \mathbb{C}') \models p_{\mathsf{t}}' .$$
  From $\mathcal{D}, R, G \models \{p'\}C\{p\}$ and $\mathsf{height}(C) = \mathcal{H}$, let $\mathbb{ws}_1' = ((0,0), |C|)$, we get:
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}_1', w_1', \mathcal{H}) \Downarrow_{\emptyset} p .$$
  Let
  $$\mathbb{ws}'' = ((0,0),0) :: ((k_s', w_1'), |C|+1) .$$
  By the co-induction hypothesis, we know
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (C; \mathsf{while}\ (B)\{C\}, \sigma) \preceq_i (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', w, \mathcal{H}+1) \Downarrow_{\xi_0'} p \wedge \neg B .$$
  Also we have $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$. Since $(\sigma, \Sigma') \models I * \mathsf{true}$, we can prove:
  $$((\sigma, \Sigma'), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True} .$$
- If $[\![B]\!]_s =$ **false**, we know $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (\mathbf{skip}, \sigma \uplus \sigma_F)$. By (SKIP) rule, let
  $$\mathbb{ws}'' = ((0,0),0) \text{ and } ws'' = (0,0) ,$$
  we know
  $$\mathcal{D}, R, G \models_{\mathsf{t}} (\mathbf{skip}, \sigma) \preceq (\mathbb{C}', \Sigma') \diamond (u, \mathbb{ws}'', ws'', w, \mathcal{H}+1) \Downarrow_{\emptyset} p \wedge \neg B .$$
  Also we have $ws'' <_{\mathcal{H}+1} ws'$ and $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$. Since $(\sigma, \Sigma') \models I * \mathsf{true}$, we can prove
  $$((\sigma, \Sigma'), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True} .$$

(5) If $((\sigma, \Sigma'), (\sigma', \Sigma''), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist u', $\mathbb{ws}''$, ws'', w'', ξ_d and ξ' such that

(a) $\mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma'') \diamond (u', \mathbb{ws}'', ws'', w'', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B$, and

(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi_0') \wedge (((\sigma, \Sigma), (\sigma', \Sigma'')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi_0' \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and

(c) if $k = 0$, then either $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$ and $w'' = w$, or $\mathbb{ws}'' = \mathbb{ws}'$ and $w'' = w$ and $\xi_d = \emptyset$; and

(d) if $k = 0$ and $(\sigma, \Sigma') \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $ws'' \leq_{\mathcal{H}+1} ws'$.

Proof: Since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know there exist u' and w_1' such that
$$((\sigma', \Sigma''), (u', w_1'), \mathbb{C}') \models p_{\mathsf{t}} \text{ and } u' \approx_k u \text{ and } k = 0 \Longrightarrow w_1' = w_1 .$$
Also we know
$$(\sigma', \Sigma'') \models J * \mathsf{true} .$$
Suppose $k_s'' = f(\sigma', \Sigma'')$. Since $J \Rightarrow (R, G \colon \mathcal{D}' \xrightarrow{f} Q)$, we can prove
$$k_s'' \leq k_s' .$$

Let

$\xi''_0 = \{\mathsf{t}'' \mid (\mathsf{t}'' \neq \mathsf{t}) \wedge ((\sigma', \Sigma'') \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}''}) * \mathsf{true})\}$ and $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi'_0) \wedge (((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$.

Since $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{wffAct}(R, \mathcal{D}')$, we can prove:

$$k = 0 \implies \xi'_0 \backslash \xi_d \subseteq \xi''_0 \ .$$

If $w'_1 = w_1$, let $w'' = w$; otherwise let $w'' = w'_1$. Thus we know $w'_1 \leq w''$. Let

$$\mathbb{ws}'' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k''_s, w'_1), 0)) \ .$$

By the co-induction hypothesis, we know

$$\mathcal{D}, R, G \models_{\mathsf{t}} (\mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}', \Sigma'') \diamond (u, \mathbb{ws}'', ws', w'', \mathcal{H}+1) \Downarrow_{\xi''_0} p \wedge \neg B \ .$$

Also we know: if $k = 0$, then $\mathbb{ws}'' \leq_{\mathcal{H}+1} \mathbb{ws}'$ and $w'' = w$.

If $k = 0$ and $\xi_d \neq \emptyset$, then there exists t' such that $\mathsf{t}' \in \xi'_0$ and $((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id}$. Since $\mathcal{D}' \leqslant \mathcal{D}$, we can prove

$$((\sigma, \Sigma'), (\sigma', \Sigma'')) \models \langle \mathcal{D}'_{\mathsf{t}'} \rangle * \mathsf{Id} \ .$$

Since $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$, we know for any $\mathsf{t}' \neq \mathsf{t}$, σ' and Σ'', if $((\sigma, \Sigma'), (\sigma', \Sigma''), 0) \models (\langle \mathcal{D}'_{\mathsf{t}'} \rangle \wedge R_{\mathsf{t}}) * \mathsf{Id}$, then $f(\sigma', \Sigma'') < k'_s$. Thus we can prove:

$$k''_s < k'_s \ .$$

Thus $\mathbb{ws}'' <_{\mathcal{H}+1} \mathbb{ws}'$ holds.

Thus we have proved (B.2).

(5) If $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist u', $\mathbb{ws}'$, ws', w', ξ_d and ξ' such that

(a) $\mathcal{D}, R, G \models_{\mathsf{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B$, and

(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \implies \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_k u$, and

(c) if $k = 0$, then either $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$ and $w' = w$, or $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi_d = \emptyset$; and

(d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $ws' \leq_{\mathcal{H}+1} ws$.

Proof: From $\mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}_1, ws_1, w_1, \mathcal{H}) \Downarrow_{\xi_1} p$, we know there exist u', $\mathbb{ws}'_1$, ws'_1, w'_1, ξ'_d and ξ'_1 such that

(A) $\mathcal{D}, R, G \models_{\mathsf{t}} (C_1, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}'_1, ws'_1, w'_1, \mathcal{H}) \Downarrow_{\xi'_1} p$, and

(B) $\xi'_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi_1) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \implies \xi_1 \backslash \xi'_d \subseteq \xi'_1)$ and $u' \approx_k u$, and

(C) if $k = 0$, then either $\mathbb{ws}'_1 <_{\mathcal{H}} \mathbb{ws}_1$ and $w'_1 = w_1$, or $\mathbb{ws}'_1 = \mathbb{ws}_1$ and $w'_1 = w_1$ and $\xi'_d = \emptyset$; and

(D) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $ws'_1 \leq_{\mathcal{H}} ws_1$.

Since $(\sigma, \Sigma) \models J * \mathsf{true}$ and $\mathsf{Sta}(J, G \vee R)$, we know

$$(\sigma', \Sigma') \models J * \mathsf{true} \ .$$

Suppose $k'_s = f(\sigma', \Sigma')$. Since $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$, we can prove

$$k'_s \leq k_s \ .$$

Also we have

$$(\sigma', \Sigma') \models Q_{\mathsf{t}} * \mathsf{true} \vee (\exists \mathsf{t}' \neq \mathsf{t}.\ \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}'}) * \mathsf{true}) \ .$$

Let

$$\begin{gathered} \xi'_0 = \{\mathsf{t}'' \mid (\mathsf{t}'' \neq \mathsf{t}) \wedge ((\sigma', \Sigma') \models \mathsf{Enabled}(\mathcal{D}'_{\mathsf{t}''}) * \mathsf{true})\}, \\ \xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\} \ \text{and} \\ \xi''_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi_0) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}. \end{gathered}$$

Since $\mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{wffAct}(R, \mathcal{D}')$, we can prove:

$$k = 0 \implies \xi_0 \backslash \xi''_d \subseteq \xi'_0 \ .$$

Then, since $\xi_1 \backslash \xi'_d \subseteq \xi'_1$, we have:

$$k = 0 \implies (\xi_0 \cup \xi_1) \backslash (\xi'_d \cup \xi''_d) \subseteq (\xi'_0 \cup \xi'_1) \ .$$

Since $\xi = \xi_0 \cup \xi_1$, we know $\xi_d = \xi'_d \cup \xi''_d$.

If $\xi''_d \neq \emptyset$, then there exists t' such that $\mathsf{t}' \in \xi_0$ and $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id}$. Since $\mathcal{D}' \leqslant \mathcal{D}$, we can prove

$$((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}'_{\mathsf{t}'} \rangle * \mathsf{Id} \ .$$

Since $J \Rightarrow (R, G\colon \mathcal{D}' \xrightarrow{f} Q)$, we know for any $\mathsf{t}' \neq \mathsf{t}$, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models (\langle \mathcal{D}'_{\mathsf{t}'} \rangle \wedge R_{\mathsf{t}}) * \mathsf{Id}$, then $f(\sigma', \Sigma') < k_s$. Then we have

$$k'_s < k_s \ .$$

Let

$$ws' = (0,0) :: \mathsf{inchead}(ws'_1, (w_1, 1)) \ .$$

We know: if $ws'_1 <_{\mathcal{H}} ws_1$, then $ws' <_{\mathcal{H}+1} ws$. If $w'_1 = w_1$, let $w' = w$; otherwise let $w' = w'_1$. Thus we know $w'_1 \leq w'$.

Suppose $\mathsf{head}(\mathbb{ws}'_1) = ((n'_1, n'_2), n'_3)$.

- If $\xi'_0 \neq \emptyset$, let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}'_1, ((k'_s, w_1), -n'_3)) \text{ and } \xi' = \xi'_0 \cup \xi'_1 .$$
Then by the co-induction hypothesis, we know
$$\mathcal{D}, R, G \models_{\mathsf{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}', ws', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B .$$
If $k = 0$, we know $\xi \backslash \xi_d \subseteq \xi'$, $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$ and $w' = w$.
For the case $k = 0$ and $\xi_d \neq \emptyset$, we know $\xi'_d \neq \emptyset$ or $\xi''_d \neq \emptyset$. If $\xi''_d \neq \emptyset$, we know $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$. If $\xi'_d \neq \emptyset$, we know $\xi_1 \neq \emptyset$ and $\mathbb{ws}'_1 <_{\mathcal{H}} \mathbb{ws}_1$. Thus $|\mathbb{ws}_1| > 1$. From the definition of $<_{\mathcal{H}}$, we can prove: $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$.
- If $\xi'_0 = \emptyset$, then we know $(\sigma', \Sigma', i) \models Q_{\mathsf{t}} * \mathsf{true}$. Let
$$\mathbb{ws}' = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}'_1, ((k'_s, w_1), 1)) \text{ and } \xi' = \xi'_0 \cup \xi'_1 = \xi'_1 .$$
By the co-induction hypothesis, we know
$$\mathcal{D}, R, G \models_{\mathsf{t}} (C_1; \mathsf{while}\ (B)\{C\}, \sigma') \preceq_i (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}', w', \mathcal{H}+1) \Downarrow_{\xi'} p \wedge \neg B .$$
If $k = 0$, we know $\xi \backslash \xi_d \subseteq \xi'$, $w' = w$ and $\mathbb{ws}'_1 \leq_{\mathcal{H}+1} \mathbb{ws}_1$.
  - If $\xi_0 \neq \emptyset$ and $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), -n_3))$, since $\xi_0 \backslash \xi''_d \subseteq \xi'_0$, we can prove
  $$\xi''_d \neq \emptyset .$$
  Then we know $k'_s < k_s$. Thus, if $k = 0$, then $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$.
  - If $(\sigma, \Sigma, i) \models Q_{\mathsf{t}} * \mathsf{true}$ and $\mathbb{ws} = ((0,0),0) :: \mathsf{inchead}(\mathbb{ws}_1, ((k_s, w_1), 1))$, we know: if $k = 0$, $\mathbb{ws}' \leq_{\mathcal{H}+1} \mathbb{ws}$. If $\xi_d \neq \emptyset$, then $\xi'_d \neq \emptyset$ or $\xi''_d \neq \emptyset$, thus $\mathbb{ws}' <_{\mathcal{H}+1} \mathbb{ws}$.

Thus we are done. □

B.2.2 The ATOM rule

Lemma 15 (ATOM-Sound). If

1. $\models_{\mathrm{SL}} [p]C[q']$;
2. $q' \Rrightarrow_k q$;
3. $(\lfloor\!\lfloor p \rfloor\!\rfloor \ltimes_k \lfloor\!\lfloor q \rfloor\!\rfloor) \Rightarrow G * \mathsf{True}$;
4. $p \vee q \Rightarrow I * \mathsf{true}$;
5. $\mathsf{Locality}(C)$;

then $\mathcal{D}, [I], G \models \{p\}\langle C \rangle\{q\}$.

Proof. Let $\mathcal{H} = \mathsf{height}(\langle C \rangle) = 1$. We know $|\langle C \rangle| = 1$. Let

$$\mathbb{ws} = ((0,0),1) \text{ and } ws = (0,1) \text{ and } \xi = \emptyset.$$

Below we prove: for any t, for any σ, Σ, u, w and $\mathbb{C}$, if $((\sigma, \Sigma), (u, w), \mathbb{C}) \models p_{\mathsf{t}}$, then

$$\mathcal{D}, [I], G \models_{\mathsf{t}} (\langle C \rangle, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} q .$$

By co-induction. Since $p \Rightarrow I * \mathsf{true}$, we know $(\sigma, \Sigma) \models I * \mathsf{true}$. From the premises, we can prove:

$$(C, \sigma) \not\longrightarrow^*_{\mathsf{t}} \textbf{abort} \text{ and } (C, \sigma) \not\longrightarrow^{\omega}_{\mathsf{t}} \cdot .$$

We only need to prove the following (1)(2)(3)(4)(5)(6).

(1) For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
Proof: It is vacantly true.
(2) If $\langle C \rangle = \textbf{skip}$, then ....
Proof: It is vacantly true.
(3) If $\langle C \rangle = \mathbf{E}[\,\textbf{return}\ E\,]$, then ....
Proof: It is vacantly true.
(4) For any σ_F, $(\langle C \rangle, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \textbf{abort}$.
Proof: By the operational semantics and $\mathsf{Locality}(C)$.
(5) For any C', σ'', σ_F and Σ_F, if $(\langle C \rangle, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$, then there exist σ', $\mathbb{C}'$, Σ', k, u', $\mathbb{ws}'$, ws', w' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
  (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
  (c) $\mathcal{D}, [I], G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} q$, and
  (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, and
  (e) either $u' <_k u$,
  or $u' = u$ and $k = 0$ and $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$,
  or $u' = u$ and $k = 0$ and $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
  (f) if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $ws' <_{\mathcal{H}} ws$.
Proof: By the operational semantics, we know C' must be **skip** and

$$(C, \sigma \uplus \sigma_F) \longrightarrow^*_{\mathsf{t}} (\textbf{skip}, \sigma'') .$$

By $\mathsf{Locality}(C)$, we know there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and

$$(C, \sigma) \longrightarrow_{\mathsf{t}}^{*} (\mathbf{skip}, \sigma') \,.$$

From $\models_{\text{SL}} [p]C[q']$, we know

$$((\sigma', \Sigma), (u, w), \mathbb{C}) \models q'_{\mathsf{t}} \,.$$

From $q' \Rrightarrow_k q$, we know: there exist u', w', $\mathbb{C}'$ and Σ' such that

(A) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and

(B) $((\sigma', \Sigma'), (u', w'), \mathbb{C}') \models q_{\mathsf{t}}$, and

(C) either $u' <_k u$, or $k = 0$ and $u' = u$ and $w' = w$.

Thus we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models (\llbracket p \rrbracket \ltimes_k \llbracket q \rrbracket) \,.$$

Since $(\llbracket p \rrbracket \ltimes_k \llbracket q \rrbracket) \Rightarrow G * \mathsf{True}$, we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True} \,.$$

By (SKIP) rule, let

$$\mathbb{ws}' = ((0, 0), 0) \;\text{ and }\; ws' = (0, 0) \,,$$

we know

$$\mathcal{D}, [I], G \models_{\mathsf{t}} (\mathbf{skip}, \sigma') \preceq (\mathbb{C}', \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\emptyset} q$$

Also we know $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $ws' <_{\mathcal{H}} ws$.

(6) If $((\sigma, \Sigma), (\sigma', \Sigma'), k') \models [I] * \mathsf{Id}$, then there exist u', $\mathbb{ws}'$, ws', w', ξ_d and ξ' such that

(a) $\mathcal{D}, R, G \models_{\mathsf{t}} (\langle C \rangle, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{ws}', ws', w', \mathcal{H}) \Downarrow_{\xi'} p \land \neg B$, and

(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \land (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k' = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$ and $u' \approx_{k'} u$, and

(c) if $k' = 0$, then either $\mathbb{ws}' <_{\mathcal{H}} \mathbb{ws}$ and $w' = w$, or $\mathbb{ws}' = \mathbb{ws}$ and $w' = w$ and $\xi_d = \emptyset$; and

(d) if $k' = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $ws' \leq_{\mathcal{H}} ws$.

Proof: We know $\sigma' = \sigma$ and $\Sigma' = \Sigma$ and $k' = 0$.

By the co-induction hypothesis, we get

$$\mathcal{D}, [I], G \models_{\mathsf{t}} (\langle C \rangle, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u, \mathbb{ws}, ws, w, \mathcal{H}) \Downarrow_{\xi} q \,.$$

Thus we are done. □

B.3 Common Simulation and Instantiations

We introduce a state-updating function Δ to describe the high-level state updates which may delay other threads. That is, at any state $\Delta_{\mathsf{t}}(\Sigma)$, another thread t' is possible to make stuttering steps.

$$\Delta \in \mathit{ThrdID} \times \mathit{Mem} \rightharpoonup \mathit{Mem}$$

Definition 16 (Common Simulation for Object)**.** $\Pi \precsim_P \Pi'$ iff there exists Δ, $\mathcal{D}$, R and G such that $\mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$ holds.

$\mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$ iff, for any $f \in \mathit{dom}(\Pi)$, for any σ and Σ, for any t, if $\Pi(f) = (x, C)$, $\Pi'(f) = (y, \mathbb{C})$ and $(\sigma, \Sigma) \models P_{\mathsf{t}} * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$, there exist two well-founded metrics $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID})$ such that

$$\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (C, \sigma) \precsim (\mathbb{C}, \Sigma) \diamond (\mathbb{M}, M) \Downarrow_{\xi} (P * \mathsf{own}(x) * \mathsf{own}(y)).$$

Here $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (C, \sigma) \precsim (\mathbb{C}, \Sigma) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ is co-inductively defined as follows.
Whenever $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (C, \sigma) \precsim (\mathbb{C}, \Sigma) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ holds, then the following hold:

(1) Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathsf{t} \notin \xi$.
For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
(2) If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \bot \Sigma_F$, there exist $\mathbb{E}$ and Σ' such that
  (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{*}_{\mathsf{t}} (\mathbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
  (b) $(\sigma, \Sigma') \models Q_{\mathsf{t}}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
  (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True}$.
(3) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$.
(4) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$ and $\Sigma \bot \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', k, $\mathbb{M}'$, M' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and
  (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n}_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
  if $k > 0$, then there exist n_1, n_2 and $\mathbb{C}''$ such that $n = n_1 + n_2 > 0$ and $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n_1}_{\mathsf{t}} (\mathbb{C}'', \Delta_{\mathsf{t}}(\Sigma) \uplus \Sigma_F)$ and $(\mathbb{C}'', \Delta_{\mathsf{t}}(\Sigma) \uplus \Sigma_F) \longrightarrow^{n_2}_{\mathsf{t}} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
  (c) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, and
  (e) either $n > 0$, or $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
  (f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle [\mathcal{D}_{\mathsf{t}}] \rangle * \mathsf{True}$, then $M' < M$.
(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that
  (a) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (C, \sigma') \precsim (\mathbb{C}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
  (c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$ and $\Sigma_F \bot \Sigma$ we have $(\mathbb{C}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}} (\mathbb{C}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F)$;
  otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
  (d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $M' \leq M$.

B.3.1 Instantiating the Common Simulation

Lemma 17 (⑦ in Fig. 18)**.** Suppose $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$. If $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, then $\mathcal{D}, R, G \models^{\emptyset} \{P\}\Pi \precsim \Gamma$.

Proof. For any $f \in \mathit{dom}(\Pi)$, for any σ and Σ, for any t, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, \mathbb{C})$ and $(\sigma, \Sigma) \models P_{\mathsf{t}} * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$, from $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, we know: there exist three well-founded metrics u, $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID})$ such that

$$\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} (P * \mathsf{own}(x) * \mathsf{own}(y)).$$

We want to prove: there exist two well-founded metric $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID})$ such that

$$\mathcal{D}, R, G \models^{\emptyset}_{\mathsf{t}} (C, \sigma) \precsim (\mathbb{C}, \Sigma) \diamond (\mathbb{M}, M) \Downarrow_{\xi} (P * \mathsf{own}(x) * \mathsf{own}(y)).$$

We only need to prove the following:

$$\text{If } \mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q, \text{ then } \mathcal{D}, R, G \models^{\emptyset}_{\mathsf{t}} (C, \sigma) \precsim (\mathbb{C}, \Sigma) \diamond ((u, \mathbb{M}), M) \Downarrow_{\xi} Q.$$

By co-induction.

(1) Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathsf{t} \notin \xi$. For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
Proof: Immediate.
(2) If $C = \mathbf{E}[\,\mathbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \bot \Sigma_F$, there exist n, $\mathbb{E}$ and Σ' such that
  (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^{n}_{\mathsf{t}} (\mathbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
  (b) $(\sigma, \Sigma') \models Q_{\mathsf{t}}$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
  (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True}$.
Proof: Immediate.
(3) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathsf{t}} \mathbf{abort}$.
Proof: Immediate.
(4) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathsf{t}} (C', \sigma'')$ and $\Sigma \bot \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', $\mathbb{M}'$, M' and ξ' such that

(a) $\sigma'' = \sigma' \uplus \sigma_F$, and
(b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{n} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
(c) $\mathcal{D}, R, G \models_{\mathsf{t}}^{\emptyset} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(d) $((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True}$, and
(e) either $n > 0$, or $\mathbb{M}' < (u, \mathbb{M})$, or $\mathbb{M}' = (u, \mathbb{M})$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $M' < M$.
Proof: Since $\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond ((u, \mathbb{M}), M) \Downarrow_{\xi} Q$, we know there exist σ', $\mathbb{C}'$, Σ', k', u', $\mathbb{M}'$, M' and ξ' such that
(A) $\sigma'' = \sigma' \uplus \sigma_F$, and
(B) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, and
(C) $\mathcal{D}, R, G \models_{\mathsf{t}} (C', \sigma') \preceq (\mathbb{C}', \Sigma') \diamond ((u', \mathbb{M}'), M') \Downarrow_{\xi'} Q$, and
(D) $((\sigma, \Sigma), (\sigma', \Sigma'), k') \models G_{\mathsf{t}} * \mathsf{True}$, and
(E) either $u' <_{k'} u$,
or $u' = u$ and $k' = 0$ and $\mathbb{M}' < \mathbb{M}$,
or $u' = u$ and $k' = 0$ and $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(F) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $M' < M$.
From (C), by the co-induction hypothesis, we know

$$\mathcal{D}, R, G \models_{\mathsf{t}}^{\emptyset} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond ((u', \mathbb{M}'), M') \Downarrow_{\xi'} Q \,.$$

From (E), by the dictionary order, we know

$$\text{either } (u', \mathbb{M}') < (u, \mathbb{M}), \text{ or } (u', \mathbb{M}') = (u, \mathbb{M}) \text{ and } \xi \neq \emptyset \text{ and } \xi \subseteq \xi'.$$

From (D), since $G \Rightarrow \lfloor G \rfloor_0$, we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models G_{\mathsf{t}} * \mathsf{True} \,.$$

(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that
(a) $\mathcal{D}, R, G \models_{\mathsf{t}}^{\emptyset} (C, \sigma') \precsim (\mathbb{C}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
(c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$ and $\Sigma_F \bot \Sigma$ we have $(\mathbb{C}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}} (\mathbb{C}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F)$;
otherwise, $\mathbb{M}' < (u, \mathbb{M})$, or $\mathbb{M}' = (u, \mathbb{M})$ and $\xi_d = \emptyset$; and
(d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $M' \leq M$.
Proof: Since $R \Rightarrow \lfloor R \rfloor_0$, we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models R_{\mathsf{t}} * \mathsf{Id} \,.$$

Then, since $\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\mathbb{C}, \Sigma) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, we know there exist u', $\mathbb{M}'$, M', ξ_d and ξ' such that
(A) $\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma') \preceq (\mathbb{C}, \Sigma') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $\xi \backslash \xi_d \subseteq \xi'$ and $u' = u$, and
(c) either $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
(d) if $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $M' \leq M$.
From (A), by the co-induction hypothesis, we know

$$\mathcal{D}, R, G \models_{\mathsf{t}}^{\emptyset} (C, \sigma') \precsim (\mathbb{C}, \Sigma') \diamond ((u', \mathbb{M}'), M') \Downarrow_{\xi'} Q \,.$$

From (C), by the dictionary order, we know

$$\text{either } (u', \mathbb{M}') < (u, \mathbb{M}), \text{ or } (u', \mathbb{M}') = (u, \mathbb{M}) \text{ and } \xi_d = \emptyset.$$

Thus we are done. □

Lemma 18 (⑧ in Fig. 18). Suppose $\mathtt{l} \notin \mathit{fv}(\mathcal{D}, R, G, P, \Pi, \Gamma)$.
If $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, then $\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models^{\Delta_1} \{P * (\mathtt{l} = 0)\}\Pi \precsim \mathsf{wr}_1(\Gamma)$. Here

$$\Delta_1(\mathsf{t}, \Sigma) \overset{\text{def}}{=} \begin{cases} \Sigma\{\mathtt{l} \leadsto \mathsf{t}\} & \text{if } \Sigma(\mathtt{l}) = 0 \\ \textit{undefined} & \text{if } \mathtt{l} \notin \textit{dom}(\Sigma) \text{ or } \Sigma(\mathtt{l}) \neq 0 \end{cases}$$

Proof. For any $f \in \textit{dom}(\Pi)$, for any σ and Σ, for any t, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, \langle \mathbb{C} \rangle; \textbf{return } \mathbb{E})$ and $(\sigma, \Sigma) \models P_{\mathsf{t}} * (\mathtt{l} = 0) * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$, we know there exists Σ_1 such that $\Sigma = \Sigma_1 \uplus \{\mathtt{l} \leadsto 0\}$ and

$$(\sigma, \Sigma_1) \models P_{\mathsf{t}} * \mathsf{own}(x) * \mathsf{own}(y) \wedge (x = y)$$

From $\mathcal{D}, R, G \models \{P\}\Pi : \Gamma$, we know: there exist three well-founded metrics u, $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\textit{ThrdID})$ such that

$$\mathcal{D}, R, G \models_{\mathsf{t}} (C, \sigma) \preceq (\langle \mathbb{C} \rangle; \textbf{return } \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} (P * \mathsf{own}(x) * \mathsf{own}(y)).$$

We want to prove: there exist two well-founded metric $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\textit{ThrdID})$ such that

$\mathsf{wr}_1(\langle\mathbb{C}\rangle) \overset{\text{def}}{=}$

```
while ( u1 >= 0 ) {
  lock l;
  unlock l;
  u1 := rand_lessthan(u1);
}
⟨ℂ⟩;
```

$\mathsf{wr}_1(\textbf{return}\ \mathbb{E}) \overset{\text{def}}{=}$

```
while ( u2 >= 0 ) {
  lock l;
  unlock l;
  u2 := rand_lessthan(u2);
}
return 𝔼;
```

$\mathsf{wr}_1'(\langle\mathbb{C}\rangle) \overset{\text{def}}{=}$

```
lock l;
unlock l;
u1 := rand_lessthan(u1);
while ( u1 >= 0 ) {
  lock l;
  unlock l;
  u1 := rand_lessthan(u1);
}
⟨ℂ⟩;
```

$\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}) \overset{\text{def}}{=}$

```
lock l;
unlock l;
u2 := rand_lessthan(u2);
while ( u2 >= 0 ) {
  lock l;
  unlock l;
  u2 := rand_lessthan(u2);
}
return 𝔼;
```

$\mathsf{wr}_1''(\langle\mathbb{C}\rangle) \overset{\text{def}}{=}$

```
unlock l;
u1 := rand_lessthan(u1);
while ( u1 >= 0 ) {
  lock l;
  unlock l;
  u1 := rand_lessthan(u1);
}
⟨ℂ⟩;
```

$\mathsf{wr}_1''(\textbf{return}\ \mathbb{E}) \overset{\text{def}}{=}$

```
unlock l;
u2 := rand_lessthan(u2);
while ( u2 >= 0 ) {
  lock l;
  unlock l;
  u2 := rand_lessthan(u2);
}
return 𝔼;
```

$\mathsf{wr}_1'''(\langle\mathbb{C}\rangle) \overset{\text{def}}{=}$

```
u1 := rand_lessthan(u1);
while ( u1 >= 0 ) {
  lock l;
  unlock l;
  u1 := rand_lessthan(u1);
}
⟨ℂ⟩;
```

$\mathsf{wr}_1'''(\textbf{return}\ \mathbb{E}) \overset{\text{def}}{=}$

```
u2 := rand_lessthan(u2);
while ( u2 >= 0 ) {
  lock l;
  unlock l;
  u2 := rand_lessthan(u2);
}
return 𝔼;
```

Figure 22. Useful notations for the abstract code wrapper.

$$\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C,\sigma) \precsim (\mathsf{wr}_1(\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}), \Sigma) \diamond (\mathbb{M}, M) \Downarrow_\xi (P * (\mathtt{l}=0) * \mathsf{own}(x) * \mathsf{own}(y)).$$

Fig. 22 gives some useful notations for the abstract code wrapper. We only need to prove the following:

1. If $\mathcal{D}, R, G \models_{\mathtt{t}} (C,\sigma) \preceq (\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_\xi Q$
   and $\Sigma = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\}$ and $0 \le u \le u_1$ and $0 \le u \le u_2$,
   then $\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C,\sigma) \precsim_i (\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma) \diamond (\mathbb{M}, M) \Downarrow_\xi (Q * (\mathtt{l}=0))$.
2. If $\mathcal{D}, R, G \models_{\mathtt{t}} (C,\sigma) \preceq (\textbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_\xi Q$
   and $\Sigma = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\}$ and $0 \le u \le u_2$,
   then $\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C,\sigma) \precsim (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma) \diamond (\mathbb{M}, M) \Downarrow_\xi (Q * (\mathtt{l}=0))$.

By co-induction. Below we use $\widehat{\mathbb{C}}$ for the abstract code in the above two goals.

(1) Suppose $\sigma = (s,h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathtt{t} \notin \xi$.
For any $\mathtt{t}' \in \xi$, we have $(\sigma, i) \models \mathsf{Enabled}(\mathcal{D}_{\mathtt{t}'}) * \mathsf{true}$.
Proof: Immediate.

(2) If $C = \mathbf{E}[\,\textbf{return}\ E\,]$, then for any Σ_F such that $\Sigma \bot \Sigma_F$, there exist $\mathbb{E}$ and Σ' such that
  (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{*} (\textbf{return}\ \mathbb{E}, \Sigma' \uplus \Sigma_F)$, and
  (b) $(\sigma, \Sigma') \models Q_{\mathtt{t}} * (\mathtt{l}=0)$ and $[\![E]\!]_{\sigma.s} = [\![\mathbb{E}]\!]_{\Sigma'.s}$, and
  (c) $((\sigma,\Sigma), (\sigma,\Sigma'), 0) \models G_{\mathtt{t}} * [\mathtt{l}=0] * \mathsf{True}$.
Proof: Immediate from the premise.

(3) For any σ_F, $(C, \sigma \uplus \sigma_F) \not\longrightarrow_{\mathtt{t}} \textbf{abort}$.
Proof: Immediate.

(4) For any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow_{\mathtt{t}} (C', \sigma'')$ and $\Sigma \bot \Sigma_F$, then there exist σ', n, $\mathbb{C}'$, Σ', k, $\mathbb{M}'$, M' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$, and

(b) $(\widehat{\mathbb{C}}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{n} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
if $k > 0$, then there exist n_1, n_2 and $\mathbb{C}''$ such that $n = n_1 + n_2 > 0$ and $(\widehat{\mathbb{C}}, \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{n_1} (\mathbb{C}'', \Delta_{\mathtt{t}}(\Sigma) \uplus \Sigma_F)$ and $(\mathbb{C}'', \Delta_{\mathtt{t}}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{n_2} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$; and
(c) $\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathtt{l} = 0))$, and
(d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathtt{t}} * [\mathtt{l} = 0] * \mathsf{True}$, and
(e) either $n > 0$, or $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(f) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathtt{t}}]\rangle * \mathsf{True}$, then $M' < M$.

Proof: For 1, from $\mathcal{D}, R, G \models_{\mathtt{t}} (C, \sigma) \preceq (\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, we know there exist σ', $\mathbb{C}'$, Σ_1', k, u', $\mathbb{M}'$, M' and ξ' such that

(A1) $\sigma'' = \sigma' \uplus \sigma_F$, and
(B1) $(\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}, \Sigma_1 \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{*} (\mathbb{C}_1', \Sigma_1' \uplus \Sigma_F)$, and
(C1) $\mathcal{D}, R, G \models_{\mathtt{t}} (C', \sigma') \preceq (\mathbb{C}_1', \Sigma_1') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(D1) $((\sigma, \Sigma_1), (\sigma', \Sigma_1'), k) \models G_{\mathtt{t}} * \mathsf{True}$, and
(E1) either $u' <_k u$,
or $u' = u$ and $k = 0$ and $\mathbb{M}' < \mathbb{M}$,
or $u' = u$ and $k = 0$ and $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(F1) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathtt{t}}]\rangle * \mathsf{True}$, then $M' < M$.

From (B1), by the operational semantics, we know
$$\text{either } \mathbb{C}_1' = (\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E}), \text{ or } \mathbb{C}_1' = (\textbf{return}\ \mathbb{E}) .$$

(i) $\mathbb{C}_1' = (\langle\mathbb{C}\rangle; \textbf{return}\ \mathbb{E})$: Let
$$\mathbb{C}' = (\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E})) \text{ and } u_1' = u' \text{ and } \Sigma' = \Sigma_1' \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1', \mathtt{u2} \rightsquigarrow u_2\} .$$
From (E1), we know: $u' \leq u$, thus $u' \leq u_2$. Then, from (C1), by the co-induction hypothesis, we know
$$\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathtt{l} = 0)) .$$
From (D1), we know
$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathtt{t}} * [\mathtt{l} = 0] * \mathsf{True} .$$

- If $k > 0$, from (E1), we know: $u' < u$. Thus $u_1' < u_1$. Let
$$\mathbb{C}'' = (\mathsf{wr}_1''(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E})) .$$
Also we know
$$\Delta_1(\mathtt{t}, \Sigma) = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow \mathtt{t}, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\} .$$
Thus we know
$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{+} (\mathbb{C}'', \Delta_1(\mathtt{t}, \Sigma) \uplus \Sigma_F) \text{ and}$$
$$(\mathbb{C}'', \Delta_1(\mathtt{t}, \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{+} (\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$
- If $k = 0$, from (E1), we know: either $\mathbb{M}' < \mathbb{M}$ or $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$. Also $u' = u$. Thus $u_1' \leq u_1$. Thus we know
$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{*} (\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$

(ii) $\mathbb{C}_1' = (\textbf{return}\ \mathbb{E})$: Let
$$\mathbb{C}' = \mathsf{wr}_1(\textbf{return}\ \mathbb{E}) \text{ and } u_2' = u' \text{ and } \Sigma' = \Sigma_1' \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2'\} .$$
From (C1), by the second goal, we know
$$\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathtt{t}}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathtt{l} = 0)) .$$
From (D1), we know
$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathtt{t}} * [\mathtt{l} = 0] * \mathsf{True} .$$
From (E1), we know $u' \leq u$, thus $u_2' \leq u_2$. From (B1), we know
$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{+} (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$
If $k > 0$, let
$$\mathbb{C}'' = (\mathsf{wr}_1''(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E})) .$$
Also we know
$$\Delta_1(\mathtt{t}, \Sigma) = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow \mathtt{t}, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\} .$$
From (B1), we know
$$(\mathsf{wr}_1'(\langle\mathbb{C}\rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{+} (\mathbb{C}'', \Delta_1(\mathtt{t}, \Sigma) \uplus \Sigma_F) \text{ and}$$
$$(\mathbb{C}'', \Delta_1(\mathtt{t}, \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{+} (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F) .$$

For 2, from $\mathcal{D}, R, G \models_{\mathtt{t}} (C, \sigma) \preceq (\textbf{return}\ \mathbb{E}, \Sigma_1) \diamond (u, \mathbb{M}, M) \Downarrow_{\xi} Q$, we know there exist σ', $\mathbb{C}'$, Σ_1', k, u', $\mathbb{M}'$, M' and ξ' such that

(A2) $\sigma'' = \sigma' \uplus \sigma_F$, and
(B2) $(\textbf{return}\ \mathbb{E}, \Sigma_1 \uplus \Sigma_F) \longrightarrow_{\mathtt{t}}^{*} (\mathbb{C}_1', \Sigma_1' \uplus \Sigma_F)$, and
(C2) $\mathcal{D}, R, G \models_{\mathtt{t}} (C', \sigma') \preceq (\mathbb{C}_1', \Sigma_1') \diamond (u', \mathbb{M}', M') \Downarrow_{\xi'} Q$, and
(D2) $((\sigma, \Sigma_1), (\sigma', \Sigma_1'), k) \models G_{\mathtt{t}} * \mathsf{True}$, and
(E2) either $u' <_k u$,
or $u' = u$ and $k = 0$ and $\mathbb{M}' < \mathbb{M}$,
or $u' = u$ and $k = 0$ and $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
(F2) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathtt{t}}]\rangle * \mathsf{True}$, then $M' < M$.

From (B2), by the operational semantics, we know $\mathbb{C}_1' = (\textbf{return}\ \mathbb{E})$. Let

$$\mathbb{C}' = \mathsf{wr}_1'(\textbf{return}\ \mathbb{E}) \ \text{ and } \ u_2' = u' \ \text{ and } \ \Sigma' = \Sigma_1' \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2'\}\ .$$

From (C2), by the co-induction hypothesis, we know

$$\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathsf{t}}^{\Delta_1} (C', \sigma') \precsim (\mathbb{C}', \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathtt{l} = 0))\ .$$

From (D2), we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * [\mathtt{l} = 0] * \mathsf{True}\ .$$

- If $k > 0$, from (E2), we know: $u' < u$. Thus $u_2' < u_2$. Let
$$\mathbb{C}'' = \mathsf{wr}_1''(\textbf{return}\ \mathbb{E})\ .$$
Also we know
$$\Delta_1(\mathsf{t}, \Sigma) = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow \mathsf{t}, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\}\ .$$
Thus we know
$$(\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{+} (\mathbb{C}'', \Delta_1(\mathsf{t}, \Sigma) \uplus \Sigma_F) \text{ and } (\mathbb{C}'', \Delta_1(\mathsf{t}, \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{+} (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F)\ .$$
- If $k = 0$, from (E2), we know: either $\mathbb{M}' < \mathbb{M}$ or $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$. Also $u' = u$. Thus $u_2' \leq u_2$. Thus we know
$$(\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma \uplus \Sigma_F) \longrightarrow_{\mathsf{t}}^{*} (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Sigma' \uplus \Sigma_F)\ .$$

(5) For any k, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * [\mathtt{l} = 0] * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that

(a) $\mathcal{D}, R * [\mathtt{l} = 0], G * [\mathtt{l} = 0] \models_{\mathsf{t}}^{\Delta_1} (C, \sigma') \precsim (\widehat{\mathbb{C}}, \Sigma') \diamond (\mathbb{M}', M') \Downarrow_{\xi'} (Q * (\mathtt{l} = 0))$, and

(b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and

(c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$ and $\Sigma_F \bot \Sigma$ we have $(\widehat{\mathbb{C}}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}} (\widehat{\mathbb{C}}, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F)$;
otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and

(d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $M' \leq M$.

Proof: Since $\{\mathtt{l}, \mathtt{u1}, \mathtt{u2}\} \cap \mathit{fv}(R) = \emptyset$, we know there exists Σ_1' such that

$$((\sigma, \Sigma_1), (\sigma', \Sigma_1'), k) \models R_{\mathsf{t}} * \mathsf{Id} \ \text{ and } \ \Sigma' = \Sigma_1' \uplus \{\mathtt{l} \rightsquigarrow 0, \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\}\ .$$

Also we know

$$\Delta_1(\mathsf{t}', \Sigma) = \Sigma_1 \uplus \{\mathtt{l} \rightsquigarrow \mathsf{t}', \mathtt{u1} \rightsquigarrow u_1, \mathtt{u2} \rightsquigarrow u_2\}\ .$$

Thus we know

$$(\mathsf{wr}_1'(\langle \mathbb{C} \rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Delta_1(\mathsf{t}', \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}} (\mathsf{wr}_1'(\langle \mathbb{C} \rangle); \mathsf{wr}_1(\textbf{return}\ \mathbb{E}), \Delta_1(\mathsf{t}', \Sigma) \uplus \Sigma_F)\ ,$$

and

$$(\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Delta_1(\mathsf{t}', \Sigma) \uplus \Sigma_F) \longrightarrow_{\mathsf{t}} (\mathsf{wr}_1'(\textbf{return}\ \mathbb{E}), \Delta_1(\mathsf{t}', \Sigma) \uplus \Sigma_F)\ .$$

By the co-induction hypothesis, we can prove the goals.

Thus we are done. □

B.4 Core Proofs: From Common Simulation to Contextual Refinement

Lemma 19 (Gray Box in Fig. 18)**.** If

1. $\mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$,
2. $\forall \mathsf{t}, \mathsf{t}'.\ \mathsf{t} \neq \mathsf{t}' \Longrightarrow G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$, $\mathsf{wffAct}(R, \mathcal{D})$, $P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})$, $P \vee \mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $I \triangleright \{R, G\}$,

then $\Pi \sqsubseteq_P \Pi'$.

Proof. From $\mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$, by Lemma 21, we get: for any C, for any t,

$$\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} \{P\}(\Pi, C) \precsim (\Pi', \mathbb{C})\{P\} .$$

(See Sec. B.4.1 for their definitions.) By Lemma 22, we know: for any n, $C_1, \ldots, C_n$,

$$\{\textstyle\bigwedge_{\mathsf{t}} P_{\mathsf{t}}\}(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n) \precsim (\textbf{let } \Pi' \textbf{ in } \mathbb{C}_1 \,\|\, \ldots \,\|\, \mathbb{C}_n) .$$

(Here the simulation $\precsim$ for the whole programs is defined in Definition 24.) By the following Lemma 25, we know

$$\{\textstyle\bigwedge_{\mathsf{t}} P_{\mathsf{t}}\}(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n) \sqsubseteq (\textbf{let } \Pi' \textbf{ in } \mathbb{C}_1 \,\|\, \ldots \,\|\, \mathbb{C}_n) .$$

(Here the fair refinement $\sqsubseteq$ is defined in Definition 23.) Thus we get: $\Pi \sqsubseteq_P \Pi'$. □

B.4.1 Lifting to Simulation for Client Threads

Definition 20 (Simulation for Thread)**.** $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} \{P\}(\Pi, C) \precsim (\Pi', \mathbb{C})\{Q\}$ iff, for any σ_c, σ and Σ, if $(\sigma, \Sigma) \models P$, there exist two well-founded metrics $\mathbb{M}$ and M and a set $\xi \in \mathscr{P}(\mathit{ThrdID})$ such that

$$\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C, (\sigma_c, \sigma, \circ)) \precsim (\Pi', \mathbb{C}, (\sigma_c, \Sigma, \circ)) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q.$$

Here $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C, (\sigma_c, \sigma, \kappa)) \precsim (\Pi', \mathbb{C}, (\Sigma_c, \Sigma, \Bbbk)) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ is co-inductively defined as follows. Whenever $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C, (\sigma_c, \sigma, \kappa)) \precsim (\Pi', \mathbb{C}, (\Sigma_c, \Sigma, \Bbbk)) \diamond (\mathbb{M}, M) \Downarrow_{\xi} Q$ holds, then the following hold:

(1) $\sigma_c = \Sigma_c$; and $(C \neq \textbf{skip}) \Rightarrow (\mathbb{C} \neq \textbf{skip})$.
(2) Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\mathtt{TIDS})$ and $\mathsf{t} \notin \xi$. For any $\mathsf{t}' \in \xi$, we have $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.
(3) If $C = \textbf{skip}$, then $\mathbb{C} = \textbf{skip}$ and $(\sigma, \Sigma) \models Q_{\mathsf{t}}$.
(4) If $(C, (\sigma_c, \sigma \uplus \sigma_F, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}$ and $\Sigma \bot \Sigma_F$, then
  $e = (\mathsf{t}, \textbf{clt}, \textbf{abort})$ and there exists $\mathbb{T}$ such that $e = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}}{}^{*}_{\mathsf{t},\Pi} \textbf{abort}$.
(5) If $(C, (\sigma_c, \sigma \uplus \sigma_F, \kappa)) \xrightarrow{e}_{\mathsf{t},\Pi} (C', (\sigma'_c, \sigma'', \kappa'))$ and $\Sigma \bot \Sigma_F$, then there exist σ', n, $\mathbb{T}$, $\mathbb{C}'$, Σ', $\Bbbk'$, k, $\mathbb{M}'$, M' and ξ' such that
  (a) $\sigma'' = \sigma' \uplus \sigma_F$;
  (b) $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}}{}^{n}_{\mathsf{t},\Pi'} (\mathbb{C}', (\sigma'_c, \Sigma' \uplus \Sigma_F, \Bbbk'))$; and
  if $k > 0$, then there exist n_1, n_2, $\mathbb{T}_1$, $\mathbb{T}_2$ and $\mathbb{C}''$ such that $n = n_1 + n_2 > 0$ and $\mathbb{T} = \mathbb{T}_1 :: \mathbb{T}_2$ and
  $(\mathbb{C}, (\Sigma_c, \Sigma \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}_1}{}^{n_1}_{\mathsf{t},\Pi'} (\mathbb{C}'', (\Sigma_c, \Delta_{\mathsf{t}}(\Sigma') \uplus \Sigma_F, \Bbbk))$ and $(\mathbb{C}'', (\Sigma_c, \Delta_{\mathsf{t}}(\Sigma') \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}_2}{}^{n_2}_{\mathsf{t},\Pi'} (\mathbb{C}', (\sigma'_c, \Sigma' \uplus \Sigma_F, \Bbbk'))$;
  and
  (c) $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$ and $(e = (\mathsf{t}, \textbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$, and
  (d) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C', (\sigma'_c, \sigma', \kappa')) \precsim (\Pi', \mathbb{C}', (\sigma'_c, \Sigma', \Bbbk')) \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (e) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_{\mathsf{t}} * \mathsf{True}$, and
  (f) either $n > 0$, or $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
  (g) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_{\mathsf{t}}]\rangle * \mathsf{True}$, then $M' < M$.
(6) For any k, σ'_c, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}} * \mathsf{Id}$, then there exist $\mathbb{M}'$, M', ξ_d and ξ' such that
  (a) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C, (\sigma'_c, \sigma', \kappa)) \precsim (\Pi', \mathbb{C}, (\sigma'_c, \Sigma', \Bbbk)) \diamond (\mathbb{M}', M') \Downarrow_{\xi'} Q$, and
  (b) $\xi_d = \{\mathsf{t}' \mid (\mathsf{t}' \in \xi) \wedge (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_{\mathsf{t}'} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi \backslash \xi_d \subseteq \xi')$, and
  (c) if $k > 0$, then for any $\mathsf{t}' \neq \mathsf{t}$, $\Sigma_F \bot \Sigma$ and Σ'_c there exists $\mathbb{T}$ such that $(\mathbb{C}, (\Sigma'_c, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F, \Bbbk)) \xrightarrow{\mathbb{T}}_{\mathsf{t},\Pi'} (\mathbb{C}, (\Sigma'_c, \Delta_{\mathsf{t}'}(\Sigma) \uplus \Sigma_F, \Bbbk))$;
  otherwise, $\mathbb{M}' < \mathbb{M}$, or $\mathbb{M}' = \mathbb{M}$ and $\xi_d = \emptyset$; and
  (d) if $k = 0$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}}) * \mathsf{true}$, then $M' \leq M$.

Lemma 21 (Lifting)**.** If $\mathit{dom}(\Pi) = \mathit{dom}(\Pi')$ and $\mathcal{D}, R, G \models^{\Delta} \{P\}\Pi \precsim \Pi'$, then for any t and C, we have $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} \{P\}(\Pi, C) \precsim (\Pi', \mathbb{C})\{P\}$.

Proof. By structural induction over C and by co-induction. □

B.4.2 Parallel Compositionality

Lemma 22 (Parallel Compositionality)**.**
If there exists R, G, $\mathcal{D}$, P and Δ such that the following hold for any $\mathsf{t} \in [1..n]$:

(1) $\mathcal{D}, R, G \models_{\mathsf{t}}^{\Delta} \{P\}(\Pi, C_{\mathsf{t}}) \precsim (\Pi', \mathbb{C}_{\mathsf{t}})\{P\}$,
(2) $\forall \mathsf{t}, \mathsf{t}'.\ \mathsf{t} \neq \mathsf{t}' \Longrightarrow G_{\mathsf{t}} \Rightarrow R_{\mathsf{t}'}$, $\mathsf{wffAct}(R, \mathcal{D})$, $P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})$, $P \vee \mathsf{Enabled}(\mathcal{D}) \Rightarrow I$, $I \rhd \{R, G\}$,

then $\{\bigwedge_{\mathsf{t}} P_{\mathsf{t}}\}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n) \precsim (\textbf{let}\ \Pi'\ \textbf{in}\ \mathbb{C}_1 \parallel \ldots \parallel \mathbb{C}_n)$.

Proof. For any σ_c, σ and Σ, if $(\sigma, \Sigma) \models (\bigwedge_{\mathsf{t}} P_{\mathsf{t}})$, from the premises and by sequential compositionality, we can prove: there exist $M_1, \ldots, M_n, \mathbb{M}_1, \ldots, \mathbb{M}_n, \xi_1, \ldots, \xi_n$ such that the following holds for any $\mathsf{t} \in [1..n]$:

$$\mathcal{D}, R, G \models_{\mathsf{t}}^{\Delta} (\Pi, (C_{\mathsf{t}}; \textbf{end}), (\sigma_c, \sigma, \circ)) \precsim (\Pi', (\mathbb{C}_{\mathsf{t}}; \textbf{end}), (\sigma_c, \Sigma, \circ)) \diamond (\mathbb{M}_{\mathsf{t}}, M_{\mathsf{t}}) \Downarrow_{\xi_{\mathsf{t}}} P\ .$$

We want to show that there exist $\mathscr{M}$ and ζ such that

$$((\textbf{let}\ \Pi\ \textbf{in}\ (C_1; \textbf{end}) \parallel \ldots \parallel (C_n; \textbf{end})), (\sigma_c, \sigma, \circledcirc)) \preceq ((\textbf{let}\ \Pi'\ \textbf{in}\ (\mathbb{C}_1; \textbf{end}) \parallel \ldots \parallel (\mathbb{C}_n; \textbf{end})), (\sigma_c, \Sigma, \circledcirc)) \diamond (\mathscr{M}, \zeta).$$

We generalize the result and prove the following (B.3):

If $(\sigma, \Sigma) \models I$ and the following holds for any $\mathsf{t} \in [1..n]$:

$$\mathcal{D}, R, G \models_{\mathsf{t}}^{\Delta} (\Pi, C_{\mathsf{t}}, (\sigma_c, \sigma \uplus \sigma_{\mathsf{t}}, \kappa_{\mathsf{t}})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}}, (\sigma_c, \Sigma \uplus \Sigma_{\mathsf{t}}, \Bbbk_{\mathsf{t}})) \diamond (\mathbb{M}_{\mathsf{t}}, M_{\mathsf{t}}) \Downarrow_{\xi_{\mathsf{t}}} P\ ,$$

then

$$((\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma \uplus (\biguplus_{\mathsf{t}} \sigma_{\mathsf{t}}), \mathcal{K})) \preceq ((\textbf{let}\ \Pi'\ \textbf{in}\ \mathbb{C}_1 \parallel \ldots \parallel \mathbb{C}_n), (\sigma_c, \Sigma \uplus (\biguplus_{\mathsf{t}} \Sigma_{\mathsf{t}}), \mathbb{K})) \diamond (\mathscr{M}, \zeta).$$

Here for any $\mathsf{t} \in [1..n]$, $\mathcal{K}(\mathsf{t}) = \kappa_{\mathsf{t}}$ and $\mathbb{K}(\mathsf{t}) = \Bbbk_{\mathsf{t}}$, and the functions $\mathscr{M}$ and ζ are defined as follows.
- $dom(\mathscr{M}) = dom(\zeta) = \mathsf{activeThrds}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n) = \{\mathsf{t} \mid (C_{\mathsf{t}} \neq \textbf{skip})\}$.
- For any $\mathsf{t} \in dom(\mathscr{M})$, we have $\mathscr{M}(\mathsf{t}) = (\mathbb{M}_{\mathsf{t}}, \{\mathsf{t}' \leadsto M_{\mathsf{t}'} \mid \mathsf{t}' \in \xi_{\mathsf{t}}\})$ and $\zeta(\mathsf{t}) = \xi_{\mathsf{t}}$.

The order $(\mathbb{M}', \mathcal{M}') < (\mathbb{M}, \mathcal{M})$ is defined as a dictionary order:

$$\begin{aligned}
&(\mathbb{M}', \mathcal{M}') < (\mathbb{M}, \mathcal{M}) \text{ iff } (\mathbb{M}' < \mathbb{M}) \vee (\mathbb{M}' = \mathbb{M}) \wedge (\mathcal{M}' < \mathcal{M})\\
&\mathcal{M}' < \mathcal{M} \text{ iff } \exists \mathsf{t}.\ (\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})) \wedge (\forall \mathsf{t}' \in dom(\mathcal{M}) \backslash \{\mathsf{t}\}.\ \mathcal{M}'(\mathsf{t}') \leq \mathcal{M}(\mathsf{t}'))\\
&\mathcal{M}' \leq \mathcal{M} \text{ iff } \forall \mathsf{t} \in dom(\mathcal{M}).\ \mathcal{M}'(\mathsf{t}) \leq \mathcal{M}(\mathsf{t})
\end{aligned}$$

Clearly $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$ is a well-founded order.

(B.3)

By co-induction. Let $W \stackrel{\text{def}}{=} (\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n)$, $\mathbb{W} \stackrel{\text{def}}{=} (\textbf{let}\ \Pi'\ \textbf{in}\ \mathbb{C}_1 \parallel \ldots \parallel \mathbb{C}_n)$, $\mathcal{S} \stackrel{\text{def}}{=} (\sigma_c, \sigma \uplus (\biguplus_{\mathsf{t}} \sigma_{\mathsf{t}}), \mathcal{K})$ and $\mathbb{S} \stackrel{\text{def}}{=} (\sigma_c, \Sigma \uplus (\biguplus_{\mathsf{t}} \Sigma_{\mathsf{t}}), \mathbb{K})$. Suppose $\sigma = (s, h)$. Then $s(\mathtt{TIDS}) = [1..n]$.

(1) $dom(\mathscr{M}) = dom(\zeta) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$, and $\forall \mathsf{t} \in dom(\zeta).\ \zeta(\mathsf{t}) \subseteq (dom(\zeta) \backslash \{\mathsf{t}\})$.
Proof: For any $\mathsf{t} \in [1..n]$, from the premise, we know: $(C_{\mathsf{t}} \neq \textbf{skip}) \Leftrightarrow (\mathbb{C}_{\mathsf{t}} \neq \textbf{skip})$. Thus $\mathsf{activeThrds}(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n) = \mathsf{activeThrds}(\textbf{let}\ \Pi'\ \textbf{in}\ \mathbb{C}_1 \parallel \ldots \parallel \mathbb{C}_n)$.
Also, $\zeta(\mathsf{t}) \subseteq s(\mathtt{TIDS}) = [1..n]$. And for any $\mathsf{t}' \in \zeta(\mathsf{t})$, we have $\mathsf{t}' \neq \mathsf{t}$ and $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$. Then we only need to prove $(C_{\mathsf{t}'} \neq \textbf{skip})$. Suppose $(C_{\mathsf{t}'} = \textbf{skip})$, then we have $(\sigma, \Sigma) \models P_{\mathsf{t}'}$. Since $P_{\mathsf{t}} \Rightarrow I$, we know $(\sigma, \Sigma) \models I$. Since $\mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) \Rightarrow I$ and $\mathsf{Precise}(\lfloor I \rfloor)$, we know $(\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$. Since $P_{\mathsf{t}'} \wedge \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) \Rightarrow \mathsf{false}$, we get $(\sigma, \Sigma) \models \mathsf{false}$, which is impossible. Thus $(C_{\mathsf{t}'} \neq \textbf{skip})$.

(2) If $(W, \mathcal{S}) \longmapsto (\textbf{skip}, \mathcal{S}')$, then there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} (\textbf{skip}, \mathbb{S}')$.
Proof: By the operational semantics we know: for any $\mathsf{t} \in [1..n]$, we have $C_{\mathsf{t}} = \textbf{skip}$. By $\mathcal{D}, R, G \models_{\mathsf{t}}^{\Delta} (\Pi, C_{\mathsf{t}}, (\sigma_c, \sigma \uplus \sigma_{\mathsf{t}}, \kappa_{\mathsf{t}})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}}, (\sigma_c, \Sigma \uplus \Sigma_{\mathsf{t}}, \Bbbk_{\mathsf{t}})) \diamond (\mathbb{M}_{\mathsf{t}}, M_{\mathsf{t}}) \Downarrow_{\xi_{\mathsf{t}}} P$, we have $\mathbb{C}_{\mathsf{t}} = \textbf{skip}$. Thus we have $(\mathbb{W}, \mathbb{S}) \longmapsto (\textbf{skip}, \mathbb{S})$.

(3) If $(W, \mathcal{S}) \overset{e}{\longmapsto} \textbf{abort}$, then there exist t and $\mathbb{T}$ such that $e = (\mathsf{t}, \textbf{clt}, \textbf{abort})$, $e = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} \textbf{abort}$.
Proof: By the operational semantics we know: there exists $\mathsf{t} \in [1..n]$ such that

$$(C_{\mathsf{t}}, (\sigma_c, \sigma \uplus (\biguplus_{\mathsf{t}} \sigma_{\mathsf{t}}), \kappa_{\mathsf{t}})) \xrightarrow{e}_{\mathsf{t},\Pi} \textbf{abort}.$$

By $\mathcal{D}, R, G \models_{\mathsf{t}}^{\Delta} (\Pi, C_{\mathsf{t}}, (\sigma_c, \sigma \uplus \sigma_{\mathsf{t}}, \kappa_{\mathsf{t}})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}}, (\sigma_c, \Sigma \uplus \Sigma_{\mathsf{t}}, \Bbbk_{\mathsf{t}})) \diamond (\mathbb{M}_{\mathsf{t}}, M_{\mathsf{t}}) \Downarrow_{\xi_{\mathsf{t}}} P$, we know $e = (\mathsf{t}, \textbf{clt}, \textbf{abort})$ and there exists $\mathbb{T}$ such that $e = \mathsf{get\_obsv}(\mathbb{T})$ and

$$(\mathbb{C}_{\mathsf{t}}, (\sigma_c, \Sigma \uplus (\biguplus_{\mathsf{t}} \Sigma_{\mathsf{t}}), \Bbbk_{\mathsf{t}})) \xrightarrow{\mathbb{T}}{}^{+}_{\mathsf{t},\Pi} \textbf{abort}.$$

Thus $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} \textbf{abort}$.

(4) If $(W, (\sigma_c, \sigma \uplus (\biguplus_{\mathsf{t}} \sigma_{\mathsf{t}}), \mathcal{K})) \overset{e}{\longmapsto} (W', (\sigma_c', \sigma'', \mathcal{K}'))$, then there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$, $\mathscr{M}'$ and ζ' such that all the following hold:
(a) $(\mathbb{W}, (\sigma_c, \Sigma \uplus (\biguplus_{\mathsf{t}} \Sigma_{\mathsf{t}}), \mathbb{K})) \overset{\mathbb{T}}{\longmapsto}{}^{*} (\mathbb{W}', \mathbb{S}')$;
(b) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \textbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
(c) $(W', (\sigma_c', \sigma'', \mathcal{K}')) \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')$;

(d) either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$,
or $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$,
or $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\zeta(\mathsf{t}) \neq \emptyset$ and $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$;

(e) for any $\mathsf{t}' \in \mathit{dom}(\mathscr{M})\backslash\{\mathsf{t}\}$, we have:
either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$,
or $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$,
or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$.

Proof: By the operational semantics, we know there exist t, C'_t and κ'_t such that

$$W' = (\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots C'_\mathsf{t} \ldots \,\|\, C_n)\ , \quad \mathsf{t} = \mathsf{tid}(e)\ ,$$
$$(C_\mathsf{t}, (\sigma_c, \sigma \uplus (\biguplus_\mathsf{t} \sigma_\mathsf{t}), \kappa_\mathsf{t})) \xrightarrow{e}_{\mathsf{t},\Pi} (C'_\mathsf{t}, (\sigma'_c, \sigma'', \kappa'_\mathsf{t}))) \quad \text{and} \quad \mathcal{K}' = \mathcal{K}\{\mathsf{t} \rightsquigarrow \kappa'_\mathsf{t}\}\ .$$

By $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C_\mathsf{t}, (\sigma_c, \sigma \uplus \sigma_\mathsf{t}, \kappa_\mathsf{t})) \precsim (\Pi', \mathbb{C}_\mathsf{t}, (\sigma_c, \Sigma \uplus \Sigma_\mathsf{t}, \mathbb{k}_\mathsf{t})) \diamond (\mathbb{M}_\mathsf{t}, M_\mathsf{t}) \Downarrow_{\xi_\mathsf{t}} P$, we know

(A) For any $\mathsf{t}' \in \xi_\mathsf{t}$, we have $\mathsf{t}' \neq \mathsf{t}$ and $(\sigma \uplus \sigma_\mathsf{t}, \Sigma \uplus \Sigma_\mathsf{t}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$.

And there exist σ''', n, $\mathbb{T}$, $\mathbb{C}'_\mathsf{t}$, Σ''', $\mathbb{k}'_\mathsf{t}$, k, $\mathbb{M}'_\mathsf{t}$, M'_t and ξ'_t such that

(B) $\sigma'' = \sigma''' \uplus (\biguplus_{\mathsf{t}' \neq \mathsf{t}} \sigma_{\mathsf{t}'})$, and

(C) $(\mathbb{C}_\mathsf{t}, (\sigma_c, \Sigma \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_\mathsf{t})) \xrightarrow{\mathbb{T}}{}^{n}_{\mathsf{t},\Pi'} (\mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma''' \uplus (\biguplus_{\mathsf{t}' \neq \mathsf{t}} \Sigma_{\mathsf{t}'}), \mathbb{k}'_\mathsf{t}))$; and
if $k > 0$, then there exist $\mathbb{C}''_\mathsf{t}$, $\mathbb{T}_1$, $\mathbb{T}_2$, n_1 and n_2 such that $\mathbb{T} = \mathbb{T}_1 :: \mathbb{T}_2$ and $n = n_1 + n_2 > 0$ and
$(\mathbb{C}_\mathsf{t}, (\sigma_c, \Sigma \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_\mathsf{t})) \xrightarrow{\mathbb{T}_1}{}^{n_1}_{\mathsf{t},\Pi'} (\mathbb{C}''_\mathsf{t}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_\mathsf{t}))$ and
$(\mathbb{C}''_\mathsf{t}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_\mathsf{t})) \xrightarrow{\mathbb{T}_2}{}^{n_2}_{\mathsf{t},\Pi'} (\mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma''' \uplus (\biguplus_{\mathsf{t}' \neq \mathsf{t}} \Sigma_{\mathsf{t}'}), \mathbb{k}'_\mathsf{t}))$;
and

(D) $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$ and $(e = (\mathsf{t}, \textbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$, and

(E) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}} (\Pi, C'_\mathsf{t}, (\sigma'_c, \sigma''', \kappa'_\mathsf{t})) \precsim (\Pi', \mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma''', \mathbb{k}'_\mathsf{t})) \diamond (\mathbb{M}'_\mathsf{t}, M'_\mathsf{t}) \Downarrow_{\xi'_\mathsf{t}} P$, and

(F) $((\sigma \uplus \sigma_\mathsf{t}, \Sigma \uplus \Sigma_\mathsf{t}), (\sigma''', \Sigma'''), k) \models G_\mathsf{t} * \mathsf{True}$, and

(G) either $n > 0$, or $\mathbb{M}'_\mathsf{t} < \mathbb{M}_\mathsf{t}$, or $\mathbb{M}'_\mathsf{t} = \mathbb{M}_\mathsf{t}$ and $\xi_\mathsf{t} \neq \emptyset$ and $\xi_\mathsf{t} \subseteq \xi'_\mathsf{t}$; and

(H) if $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle[\mathcal{D}_\mathsf{t}]\rangle * \mathsf{True}$, then $M'_\mathsf{t} < M_\mathsf{t}$.

Since $(\sigma, \Sigma) \models I$ and $I \rhd \{R, G\}$, we know there exist σ', σ'_t, Σ' and Σ'_t such that

$$\sigma''' = \sigma' \uplus \sigma'_\mathsf{t}, \quad \Sigma''' = \Sigma' \uplus \Sigma'_\mathsf{t}, \quad (\sigma', \Sigma') \models I \quad \text{and} \quad ((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_\mathsf{t}.$$

For any $\mathsf{t}' \neq \mathsf{t}$, since $G_\mathsf{t} \Rightarrow R_{\mathsf{t}'}$, we have $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models R_{\mathsf{t}'}$. Thus

$$((\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}), (\sigma' \uplus \sigma_{\mathsf{t}'}, \Sigma' \uplus \Sigma_{\mathsf{t}'}), k) \models R_{\mathsf{t}'} * \mathsf{Id}.$$

By $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}'} (\Pi, C_{\mathsf{t}'}, (\sigma_c, \sigma \uplus \sigma_{\mathsf{t}'}, \kappa_{\mathsf{t}'})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}'}, (\sigma_c, \Sigma \uplus \Sigma_{\mathsf{t}'}, \mathbb{k}_{\mathsf{t}'})) \diamond (\mathbb{M}_{\mathsf{t}'}, M_{\mathsf{t}'}) \Downarrow_{\xi_{\mathsf{t}'}} P$, we know

(I) For any $\mathsf{t}'' \in \xi_{\mathsf{t}'}$, we have $\mathsf{t}'' \neq \mathsf{t}'$ and $(\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}''}) * \mathsf{true}$.

And there exist $\mathbb{M}'_{\mathsf{t}'}$, $M'_{\mathsf{t}'}$, ξ_d and $\xi'_{\mathsf{t}'}$ such that

(J) $\mathcal{D}, R, G \models^{\Delta}_{\mathsf{t}'} (\Pi, C_{\mathsf{t}'}, (\sigma'_c, \sigma' \uplus \sigma_{\mathsf{t}'}, \kappa_{\mathsf{t}'})) \precsim (\Pi', \mathbb{C}_{\mathsf{t}'}, (\sigma'_c, \Sigma' \uplus \Sigma_{\mathsf{t}'}, \mathbb{k}_{\mathsf{t}'})) \diamond (\mathbb{M}'_{\mathsf{t}'}, M'_{\mathsf{t}'}) \Downarrow_{\xi'_{\mathsf{t}'}} P$, and

(K) $\xi_d = \{\mathsf{t}'' \mid (\mathsf{t}'' \in \xi_{\mathsf{t}'}) \wedge (((\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}), (\sigma' \uplus \sigma_{\mathsf{t}'}, \Sigma' \uplus \Sigma_{\mathsf{t}'})) \models \langle \mathcal{D}_{\mathsf{t}''} \rangle * \mathsf{Id})\}$ and $(k = 0 \Longrightarrow \xi_{\mathsf{t}'} \backslash \xi_d \subseteq \xi'_{\mathsf{t}'})$, and

(L) if $k > 0$, then there exists $\mathbb{T}'$ such that $(\mathbb{C}_{\mathsf{t}'}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_{\mathsf{t}'})) \xrightarrow{\mathbb{T}'}_{\mathsf{t}',\Pi'} (\mathbb{C}_{\mathsf{t}'}, (\sigma_c, \Delta_\mathsf{t}(\Sigma) \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_{\mathsf{t}'}))$,
otherwise, $\mathbb{M}'_{\mathsf{t}'} < \mathbb{M}_{\mathsf{t}'}$, or $\mathbb{M}'_{\mathsf{t}'} = \mathbb{M}_{\mathsf{t}'}$ and $\xi_d = \emptyset$; and

(M) if $k = 0$ and $(\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$, then $M'_{\mathsf{t}'} \leq M_{\mathsf{t}'}$.

From (C), we know

$$(\mathbb{C}_\mathsf{t}, (\sigma_c, \Sigma \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{k}_\mathsf{t})) \xrightarrow{\mathbb{T}}{}^{n}_{\mathsf{t},\Pi'} (\mathbb{C}'_\mathsf{t}, (\sigma'_c, \Sigma' \uplus \Sigma'_\mathsf{t} \uplus (\biguplus_{\mathsf{t}' \neq \mathsf{t}} \Sigma_{\mathsf{t}'}), \mathbb{k}'_\mathsf{t}))\ .$$

Let $\mathbb{W}' = (\textbf{let}\ \Pi'\ \textbf{in}\ \mathbb{C}_1 \,\|\, \ldots \mathbb{C}'_\mathsf{t} \ldots \,\|\, \mathbb{C}_n)$ and $\mathbb{S}' = (\sigma'_c, \Sigma' \uplus \Sigma'_\mathsf{t} \uplus (\biguplus_{\mathsf{t}' \neq \mathsf{t}} \Sigma_{\mathsf{t}'}), \mathbb{K}\{\mathsf{t} \rightsquigarrow \mathbb{k}'_\mathsf{t}\})$. With (L), we know there exist $\mathbb{T}$ and n such that

$$(\mathbb{W}, (\sigma_c, \Sigma \uplus (\biguplus_\mathsf{t} \Sigma_\mathsf{t}), \mathbb{K})) \overset{\mathbb{T}}{\longmapsto}{}^{n} (\mathbb{W}', \mathbb{S}')\ .$$

Also, for any $\mathsf{t}' \neq \mathsf{t}$, we have: either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathbb{M}'_{\mathsf{t}'} < \mathbb{M}_{\mathsf{t}'}$, or $\mathbb{M}'_{\mathsf{t}'} = \mathbb{M}_{\mathsf{t}'}$ and $\xi_d = \emptyset$.
From (G), we know: either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$, or $\mathbb{M}'_\mathsf{t} < \mathbb{M}_\mathsf{t}$, or $\mathbb{M}'_\mathsf{t} = \mathbb{M}_\mathsf{t}$ and $\xi_\mathsf{t} \neq \emptyset$ and $\xi_\mathsf{t} \subseteq \xi'_\mathsf{t}$.
Besides, we have

$$k > 0 \implies (\mathsf{t} \in \mathsf{tidset}(\mathbb{T})) \wedge (\mathsf{t}' \in \mathsf{tidset}(\mathbb{T}))\ .$$

Define the functions $\mathscr{M}'$ and ζ' as follows.

- $\mathit{dom}(\mathscr{M}') = \mathit{dom}(\zeta') = \mathsf{activeThrds}(W')$.
- For any $\mathsf{t} \in \mathit{dom}(\mathscr{M}')$, we have $\mathscr{M}'(\mathsf{t}) = (\mathbb{M}'_\mathsf{t}, \{\mathsf{t}' \rightsquigarrow M'_{\mathsf{t}'} \mid \mathsf{t}' \in \xi'_\mathsf{t}\})$ and $\zeta'(\mathsf{t}) = \xi'_\mathsf{t}$.

Then by the co-induction hypothesis, we know

$$(W', (\sigma'_c, \sigma'', \mathcal{K}')) \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')\ .$$

For the thread t, suppose $\mathsf{t} \notin \mathsf{tidset}(\mathbb{T})$, then $k = 0$. Then,

- If $\mathbb{M}'_{\mathsf{t}} < \mathbb{M}_{\mathsf{t}}$, then $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$.
- If $\mathbb{M}'_{\mathsf{t}} = \mathbb{M}_{\mathsf{t}}$ and $\xi_{\mathsf{t}} \neq \emptyset$ and $\xi_{\mathsf{t}} \subseteq \xi'_{\mathsf{t}}$, we know $\zeta(\mathsf{t}) \neq \emptyset$ and $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$. We only need to show the following (B.4):
$$\{\mathsf{t}' \leadsto M'_{\mathsf{t}'} \mid \mathsf{t}' \in \xi'_{\mathsf{t}}\} \;\leq\; \{\mathsf{t}' \leadsto M_{\mathsf{t}'} \mid \mathsf{t}' \in \xi_{\mathsf{t}}\} \tag{B.4}$$
From (A), since $\forall \mathsf{t}'.\ \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) \Rightarrow I$ and $\mathsf{Precise}(I)$, we know: for any $\mathsf{t}' \in \xi_{\mathsf{t}}$,
$$\mathsf{t}' \neq \mathsf{t} \quad \text{and} \quad (\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})\ .$$
Thus $(\sigma \uplus \sigma_{\mathsf{t}'}, \Sigma \uplus \Sigma_{\mathsf{t}'}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'}) * \mathsf{true}$. From (M), we have $M'_{\mathsf{t}'} \leq M_{\mathsf{t}'}$. Thus (B.4) holds.

For any $\mathsf{t}' \in dom(\mathscr{M})\backslash\{\mathsf{t}\}$, suppose $\mathsf{t}' \notin \mathsf{tidset}(\mathbb{T})$, then $k = 0$. Then,

- If $\mathbb{M}'_{\mathsf{t}'} < \mathbb{M}_{\mathsf{t}'}$, then $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$.
- If $\mathbb{M}'_{\mathsf{t}'} = \mathbb{M}_{\mathsf{t}'}$, $\xi_d = \emptyset$ and $\mathsf{t} \notin \xi_{\mathsf{t}'}$, from (K), we know $\xi_{\mathsf{t}'} \subseteq \xi'_{\mathsf{t}'}$. Thus $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$. We only need to show the following (B.5):
$$\{\mathsf{t}'' \leadsto M'_{\mathsf{t}''} \mid \mathsf{t}'' \in \xi'_{\mathsf{t}'}\} \;\leq\; \{\mathsf{t}'' \leadsto M_{\mathsf{t}''} \mid \mathsf{t}'' \in \xi_{\mathsf{t}'}\} \tag{B.5}$$
From (I), we know: for any $\mathsf{t}'' \in \xi_{\mathsf{t}'}$,
$$\mathsf{t}'' \neq \mathsf{t}' \quad \text{and} \quad (\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}''})\ .$$
Thus $(\sigma \uplus \sigma_{\mathsf{t}''}, \Sigma \uplus \Sigma_{\mathsf{t}''}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}''}) * \mathsf{true}$. From (M), we have $M'_{\mathsf{t}''} \leq M_{\mathsf{t}''}$. Thus (B.5) holds.
- If $\mathbb{M}'_{\mathsf{t}'} = \mathbb{M}_{\mathsf{t}'}$, $\xi_d = \emptyset$ and $\mathsf{t} \in \xi_{\mathsf{t}'}$, since $\mathsf{wffAct}(R, \mathcal{D})$ and $k = 0$, we know
$$((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle [\mathcal{D}_{\mathsf{t}}] \rangle\ .$$
From (H), we know
$$M'_{\mathsf{t}} < M_{\mathsf{t}}.$$
We only need to show the following (B.6):
$$\{\mathsf{t}'' \leadsto M'_{\mathsf{t}''} \mid \mathsf{t}'' \in \xi'_{\mathsf{t}'}\} \;<\; \{\mathsf{t}'' \leadsto M_{\mathsf{t}''} \mid \mathsf{t}'' \in \xi_{\mathsf{t}'}\} \tag{B.6}$$
For any $\mathsf{t}'' \in \xi_{\mathsf{t}'}\backslash\{\mathsf{t}\}$, from (I), we know:
$$\mathsf{t}'' \neq \mathsf{t}' \quad \text{and} \quad (\sigma, \Sigma) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}''})\ .$$
Thus $(\sigma \uplus \sigma_{\mathsf{t}''}, \Sigma \uplus \Sigma_{\mathsf{t}''}) \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}''}) * \mathsf{true}$. From (M), we have $M'_{\mathsf{t}''} \leq M_{\mathsf{t}''}$. Thus (B.6) holds.

Thus we are done. □

B.4.3 Simulation for Whole Programs and Fair Refinement

Definition 23 (Fair refinement)**.** $\{P\}W \sqsubseteq \mathbb{W}$ iff
$$\forall \sigma_c, \sigma, \Sigma.\ (\sigma, \Sigma) \models P \implies \mathcal{O}_{f\omega}[\![W, (\sigma_c, \sigma, \circledcirc)]\!] \subseteq \mathcal{O}_{f\omega}[\![\mathbb{W}, (\sigma_c, \Sigma, \circledcirc)]\!]\ .$$

Definition 24 (Simulation for the whole program)**.** $\{P\}W \precsim \mathbb{W}$ iff, for any σ_c, σ and Σ, if $(\sigma, \Sigma) \models P$, there exist $\mathscr{M} \in \mathit{ThrdID} \rightharpoonup \mathit{Metric}$ and $\zeta \in \mathit{ThrdID} \rightharpoonup \mathscr{P}(\mathit{ThrdID})$ such that
$$(\lfloor W \rfloor, (\sigma_c, \sigma, \circledcirc)) \preceq (\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc)) \diamond (\mathscr{M}, \zeta).$$

Here $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$ is co-inductively defined as follows.
Whenever $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$ holds, then the following hold:

(1) $dom(\mathscr{M}) = dom(\zeta) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$, and $\forall \mathsf{t} \in dom(\zeta).\ \zeta(\mathsf{t}) \subseteq (dom(\zeta)\backslash\{\mathsf{t}\})$.

(2) If $(W, \mathcal{S}) \longmapsto (\mathbf{skip}, \mathcal{S}')$, then
there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}}{}^{+} (\mathbf{skip}, \mathbb{S}')$.

(3) If $(W, \mathcal{S}) \xrightarrow{e} \mathbf{abort}$, then
there exist t and $\mathbb{T}$ such that $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$, $e = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}}{}^{+} \mathbf{abort}$.

(4) If $(W, \mathcal{S}) \xrightarrow{e} (W', \mathcal{S}')$, then
there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$, $\mathscr{M}'$ and ζ' such that all the following hold:
  (a) $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}}{}^{*} (\mathbb{W}', \mathbb{S}')$;
  (b) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \mathbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
  (c) $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')$;
  (d) either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$,
  or $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$,
  or $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\zeta(\mathsf{t}) \neq \emptyset$ and $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$;
  (e) for any $\mathsf{t}' \in dom(\mathscr{M})\backslash\{\mathsf{t}\}$, we have:
  either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$,
  or $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$,
  or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$.

Lemma 25 (Simulation for whole program ensures fair refinement)**.**
If $\{P\}W \precsim \mathbb{W}$ and $|W| = |\mathbb{W}|$, then $\{P\}W \sqsubseteq \mathbb{W}$.

To prove Lemma 25, we introduce the notion of scheduler $\mathfrak{T}$, which is similar to the event trace T but contains more information such that it can uniquely determine an execution. That is, each step is deterministic given the code, the state, and the scheduler. Note that the event trace T is not sufficient to determine a unique execution if we allow non-deterministic instructions such as `x := rand()`.

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots C_{\mathsf{t}} \ldots \,\|\, C_n)|_{\mathsf{t}} \overset{\text{def}}{=} C_{\mathsf{t}}$$

$$\mathsf{activeThrds}(W) \overset{\text{def}}{=} \{\mathsf{t} \mid \exists C.\, (W|_{\mathsf{t}} = C) \wedge (C \neq \textbf{skip})\}$$

$$\mathsf{tidset}(T) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } T = \epsilon \\ \{\mathsf{tid}(e)\} \cup \mathsf{tidset}(T') & \text{if } T = e :: T' \end{cases}$$

Figure 23. Auxiliary definitions.

$$\begin{array}{rrcl} (\mathit{SchEvt}) & \iota & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \textbf{ret}, n) \mid (\mathsf{t}, \textbf{obj}, \mathit{info}) \mid (\mathsf{t}, \textbf{obj}, \textbf{abort}) \\ & & \mid & (\mathsf{t}, \textbf{out}, n, \mathit{info}) \mid (\mathsf{t}, \textbf{clt}, \mathit{info}) \mid (\mathsf{t}, \textbf{clt}, \textbf{abort}) \\ & & \mid & (\mathsf{t}, \textbf{term}) \mid (\textbf{spawn}, n) \\ (\mathit{Sched}) & \mathfrak{T} & ::= & \epsilon \mid \iota :: \mathfrak{T} \qquad \text{(co-inductive)} \end{array}$$

Here *info* denotes the additional information that can uniquely determine a step. For instance, it could be the value written to `x` for the non-deterministic instruction `x := rand()`. We can even record the initial and final states of the step if necessary.

We strengthen the labelled transition system $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ in Fig. 5 to $(W, \mathcal{S}) \overset{\iota}{\longmapsto} (W', \mathcal{S}')$. The definition of the strengthened transition system is similar to the original one and omitted here. Then $(W, \mathcal{S}) \overset{\mathfrak{T}}{\longmapsto}{}^{*} (W', \mathcal{S}')$ represents a zero or multi-step execution of $(W, \mathcal{S})$ that leads to $(W', \mathcal{S}')$ under the scheduler $\mathfrak{T}$. $(W, \mathcal{S}) \overset{\mathfrak{T}}{\longmapsto}{}^{\omega} \cdot$ represents an infinite execution with the infinite scheduler $\mathfrak{T}$. We use $e = \lfloor \iota \rfloor$ to remove the additional information in ι, and $T = \lfloor \mathfrak{T} \rfloor$ to remove the additional information in each event in $\mathfrak{T}$. We also overload all the predicates over T to be defined over $\mathfrak{T}$. For instance, $\mathsf{get\_obsv}(\mathfrak{T})$ gives us an observable "scheduler" $\mathfrak{T}_o$, which is the subsequence of $\mathfrak{T}$ consisting of externally observable events only.

To facilitate the proofs, we give alternative definitions of $\mathcal{O}_{f\omega}$. Fig. 24 shows the alternative co-inductive definitions for generating observable event traces of complete fair executions. We prove they are equivalent to the original definitions, as shown in the following lemmas. Their proofs are given later.

Lemma 26. $(T_o \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]) \iff (\exists \mathfrak{T}, \mathfrak{T}_o.\, (\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(\lfloor W \rfloor, \mathcal{S}, \mathfrak{T}_o)) \wedge (\lfloor \mathfrak{T}_o \rfloor = T_o))$.

Lemma 27. $(\exists \mathfrak{T}.\, \mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)) \iff \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$.

Lemma 28. If $\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$, then $\mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$.

We also define a simulation between whole programs under an explicit scheduler $\mathfrak{T}$ at the low level, $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, as a bridge that relates the simulation $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$ in Definition 24 to the fair refinement.

Definition 29 (Simulation for the whole program with fixed scheduling at the low level)**.**
$\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$ is co-inductively defined as follows. Here $\mathcal{M} \in \mathit{ThrdID} \rightharpoonup \mathit{Metric}$.
Whenever $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$ holds, then the following hold:

(1) $\mathit{dom}(\mathcal{M}) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$.
(2) If $(W, \mathcal{S}) \longmapsto (\textbf{skip}, \mathcal{S}')$ and $\mathfrak{T} = \epsilon$, then
 there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} (\textbf{skip}, \mathbb{S}')$.
(3) If $(W, \mathcal{S}) \overset{\iota}{\longmapsto} \textbf{abort}$ and $\mathfrak{T} = \iota :: \epsilon$, then
 there exist t and $\mathbb{T}$ such that $\lfloor \iota \rfloor = (\mathsf{t}, \textbf{clt}, \textbf{abort})$, $\lfloor \iota \rfloor = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{+} \textbf{abort}$.
(4) If $(W, \mathcal{S}) \overset{\iota}{\longmapsto} (W', \mathcal{S}')$ and $\mathfrak{T} = \iota :: \mathfrak{T}'$, then
 there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$ and $\mathcal{M}'$ such that all the following hold:
 (a) $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^{*} (\mathbb{W}', \mathbb{S}')$;
 (b) $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, $\mathsf{get\_obsv}(\lfloor \iota \rfloor) = \mathsf{get\_obsv}(\mathbb{T})$, $(\lfloor \iota \rfloor = (\mathsf{t}, \textbf{term})) \Rightarrow (\lfloor \iota \rfloor = \mathsf{last}(\mathbb{T}))$;
 (c) $\mathfrak{T}' \models (W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$;
 (d) for any $\mathsf{t}' \in \mathit{dom}(\mathcal{M})$, either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.

Lemma 30. For any $\mathfrak{T}$, W, $\mathcal{S}$, $\mathfrak{T}_o$, $\mathbb{W}$, $\mathbb{S}$, $\mathscr{M}$ and ξ, if $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$ and $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, then there exists $\mathcal{M}$ such that $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$.

Lemma 31. If $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$ and $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, then $\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(\mathbb{W}, \mathbb{S}, \mathfrak{T}_o)$.

The proofs of the above lemmas are given later. With the auxiliary simulation with an explicit scheduler and these useful lemmas, we can finish the proof of Lemma 25.

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)} \qquad \frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (\textbf{skip},\_) \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)}$$

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \epsilon \qquad \mathfrak{T}' \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W',\mathcal{S}',\epsilon) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T})}{\mathfrak{T} :: \mathfrak{T}' \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\epsilon)}$$

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \iota :: \mathfrak{T}_o \qquad \mathfrak{T}' \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W',\mathcal{S}',\mathfrak{T}'_o) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T})}{\mathfrak{T} :: \mathfrak{T}' \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},e :: \mathfrak{T}_o :: \mathfrak{T}'_o)}$$

(a) with an explicit scheduler

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)} \qquad \frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (\textbf{skip},\_) \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)}$$

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \epsilon \qquad \mathcal{O}^{\mathsf{co}}_{f\omega}(W',\mathcal{S}',\epsilon) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T})}{\mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},\epsilon)}$$

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \iota :: \mathfrak{T}_o \qquad \mathcal{O}^{\mathsf{co}}_{f\omega}(W',\mathcal{S}',\mathfrak{T}'_o) \qquad \mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T})}{\mathcal{O}^{\mathsf{co}}_{f\omega}(W,\mathcal{S},e :: \mathfrak{T}_o :: \mathfrak{T}'_o)}$$

(b) the scheduler is implicit

$$\frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} \textbf{abort} \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)} \qquad \frac{(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (\textbf{skip},\_) \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o}{\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W,\mathcal{S},\mathfrak{T}_o)}$$

$$\frac{\begin{array}{c}(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{*} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \epsilon \qquad \mathcal{M}' \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W',\mathcal{S}',\epsilon) \\ \mathit{dom}(\mathcal{M}) = \mathsf{activeThrds}(W) \neq \emptyset \qquad \forall \mathsf{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}).\ \mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})\end{array}}{\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W,\mathcal{S},\epsilon)}$$

$$\frac{\begin{array}{c}(W,\mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+} (W',\mathcal{S}') \qquad \mathsf{get\_obsv}(\mathfrak{T}) = \iota :: \mathfrak{T}_o \qquad \mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W',\mathcal{S}',\mathfrak{T}'_o) \\ \mathit{dom}(\mathcal{M}) = \mathsf{activeThrds}(W) \qquad \forall \mathsf{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}).\ \mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})\end{array}}{\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W,\mathcal{S},e :: \mathfrak{T}_o :: \mathfrak{T}'_o)}$$

(c) with the metric mapping $\mathcal{M} \in \mathit{ThrdID} \rightharpoonup \mathit{Metric}$

Figure 24. Co-inductive definitions for generating observable event traces of complete fair executions.

Proof of Lemma 25. By Lemmas 26 and 27, we only need to show: for any σ_c, σ, Σ, i, $\mathfrak{T}$ and $\mathfrak{T}_o$, if $(\sigma, \Sigma, i) \models P$ and $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(\lfloor W \rfloor, (\sigma_c, \sigma, \circledcirc), \mathfrak{T}_o)$, then $\mathcal{O}^{\mathsf{co}}_{f\omega}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \mathfrak{T}_o)$.

From $\{P\}W \precsim \mathbb{W}$, we know there exists $\mathscr{M}$ and ζ such that

$$(\lfloor W \rfloor, (\sigma_c, \sigma, \circledcirc)) \preceq (\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc)) \diamond (\mathscr{M}, \zeta).$$

By Lemma 30, we know there exists $\mathcal{M}$ such that

$$\mathfrak{T} \models (\lfloor W \rfloor, (\sigma_c, \sigma, \circledcirc)) \preceq (\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc)) \diamond \mathcal{M}.$$

Then by Lemma 31, we know

$$\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \mathfrak{T}_o).$$

By Lemma 28, we get

$$\mathcal{O}^{\mathsf{co}}_{f\omega}(\lfloor \mathbb{W} \rfloor, (\sigma_c, \Sigma, \circledcirc), \mathfrak{T}_o).$$

Thus we are done. □

Proof of Lemma 26. $T_o \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ can be unfolded as follows.

$$\begin{array}{lll} T_o \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!] & \text{iff} & \exists T.\ ((\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{+} \textbf{abort}) \wedge (T_o = \mathsf{get\_obsv}(T)) \\ & & \quad \vee ((\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{+} (\textbf{skip}, \_)) \wedge (T_o = \mathsf{get\_obsv}(T)) \\ & & \quad \vee ((\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot) \wedge (T_o = \mathsf{get\_obsv}(T)) \\ & & \qquad \wedge (\forall \mathsf{t} \in \mathsf{activeThrds}(\lfloor W \rfloor).\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})) \end{array}$$

We define $(\mathfrak{T}_o, \mathfrak{T}) \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ as follows.

$$\begin{array}{lll} (\mathfrak{T}_o, \mathfrak{T}) \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!] & \text{iff} & ((W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^{+} \textbf{abort}) \wedge (\mathfrak{T}_o = \mathsf{get\_obsv}(\mathfrak{T})) \\ & & \vee ((W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^{+} (\textbf{skip}, \_)) \wedge (\mathfrak{T}_o = \mathsf{get\_obsv}(\mathfrak{T})) \\ & & \vee ((W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^{\omega} \cdot) \wedge (\mathfrak{T}_o = \mathsf{get\_obsv}(\mathfrak{T})) \\ & & \quad \wedge (\forall \mathsf{t} \in \mathsf{activeThrds}(W).\ |(\mathfrak{T}|_{\mathsf{t}})| = \omega \vee \mathsf{last}(\mathfrak{T}|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})) \end{array}$$

Then $(T_o \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]) \iff (\exists \mathfrak{T}_o, \mathfrak{T}.\ ((\mathfrak{T}_o, \mathfrak{T}) \in \mathcal{O}_{f\omega}[\![(\lfloor W \rfloor), \mathcal{S}]\!]) \wedge (\lfloor \mathfrak{T}_o \rfloor = T_o))$. Below we prove

$$(\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)) \iff ((\mathfrak{T}_o, \mathfrak{T}) \in \mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]) \tag{B.7}$$

- $\Longrightarrow$: We have two cases:
  - $|\mathfrak{T}| \neq \omega$:
    Proof: By induction over $|\mathfrak{T}|$, and then by inversion over $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$.
  - $|\mathfrak{T}| = \omega$: We prove $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^{\omega} \cdot$, $\mathfrak{T}_o = \mathsf{get\_obsv}(\mathfrak{T})$ and $\forall \mathsf{t} \in \mathsf{activeThrds}(W).\ |(\mathfrak{T}|_{\mathsf{t}})| = \omega \vee \mathsf{last}(\mathfrak{T}|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})$.
    - We prove $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^{\omega} \cdot$ by co-induction.
    - We prove $\mathfrak{T}_o = \mathsf{get\_obsv}(\mathfrak{T})$ by co-induction.
    - Let $\mathsf{termThrds}(\mathfrak{T}) \overset{\text{def}}{=} \{\mathsf{t} \mid \mathsf{last}(\mathfrak{T}|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})\}$. For any $\mathsf{t} \in \mathsf{activeThrds}(W) \backslash \mathsf{termThrds}(\mathfrak{T})$, we prove $|(\mathfrak{T}|_{\mathsf{t}})| = \omega$ by co-induction.
- $\Longleftarrow$: By co-induction.

□

Proof of Lemma 27. By co-induction. □

Proof of Lemma 28. We want to prove the following:

$$\begin{array}{l}\forall W, \mathcal{S}, \mathfrak{T}_o.\\ (\,\exists \mathcal{M}.\ (\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o))\,) \implies \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)\end{array}$$

By co-induction.

$$\textsf{Co-induction Principle: } \forall x.\ (\exists S.\ S \subseteq F(S) \wedge x \in S) \implies x \in \text{gfp } F$$

Figure 24 defines F and gfp F (i.e., $\mathcal{O}^{\mathsf{co}}_{f\omega}$ at the middle part of the figure). Let

$$S \stackrel{\text{def}}{=} \{(W, \mathcal{S}, \mathfrak{T}_o) \mid \exists \mathcal{M}.\ (\mathcal{M} \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o))\}.$$

So from the co-induction principle, we only need to prove:

$$S \subseteq F(S), \text{ i.e., } \forall W, \mathcal{S}, \mathfrak{T}_o.\ (W, \mathcal{S}, \mathfrak{T}_o) \in S \implies (W, \mathcal{S}, \mathfrak{T}_o) \in F(S)\,.$$

By unfolding S and by inversion over $\mathcal{O}^{\mathsf{m}}_{f\omega}$, we have the following cases.

1. $(W, \mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+}$ **abort** and $\mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o$:
   From the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
2. $(W, \mathcal{S}) \xmapsto{\mathfrak{T}}{}^{+}$ (**skip**, _) and $\mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o$:
   From the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
3. $\mathfrak{T}_o = \epsilon$, $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x}{}^{*} (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\mathfrak{T}_x) = \epsilon$, $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon))$, $\mathit{dom}(\mathcal{M}) = \mathsf{activeThrds}(W) \neq \emptyset$ and $\forall \mathsf{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t})$:
   We want to prove $(\mathfrak{T}, W, \mathcal{S}, \epsilon) \in F(S)$. We have two cases:
   (a) $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$:
   Since $\mathsf{activeThrds}(W) \neq \emptyset$, we know $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x}{}^{+} (W_y, \mathcal{S}_y)$. From $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon))$, we know $(W_y, \mathcal{S}_y, \epsilon) \in S$. From the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \epsilon) \in F(S)$.
   (b) $\mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x) \neq \emptyset$:
   By transfinite induction over the metric $(\mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x), \mathcal{M}_y)$.
   $$\textsf{Transfinite Induction Principle: } (\forall M.\ (\forall M'.\ M' < M \Longrightarrow P(M')) \Longrightarrow P(M)\,) \implies \forall M.P(M)$$
   The order over the metrics, $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$, is defined as follows.
   $$(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1) \text{ iff } (\xi_2 \subset \xi_1) \vee (\xi_2 = \xi_1 \neq \emptyset) \wedge (\forall \mathsf{t} \in \xi_2.\ \mathcal{M}_2(\mathsf{t}) < \mathcal{M}_1(\mathsf{t}))$$
   It is clear that $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$ is a well-founded order.
   We view our goal as $\forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y.P(\mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y)$, which is reformulated as follows.
   $$\begin{array}{l}\forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y.\\ \quad \forall W_y, \mathcal{S}_y.\\ \quad ((W, \mathcal{S}) \xmapsto{\mathfrak{T}_x}{}^{*} (W_y, \mathcal{S}_y)) \wedge (\mathsf{get\_obsv}(\mathfrak{T}_x) = \epsilon)\\ \quad \wedge\, (\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon)) \wedge (\forall \mathsf{t} \in \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t}))\\ \quad \Longrightarrow\\ \quad (W, \mathcal{S}, \epsilon) \in F(S)\end{array}$$
   By the transfinite induction principle, we only need to prove
   $$\begin{array}{l}\forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y.\\ \quad (\forall \mathcal{M}', \mathfrak{T}'_x, \mathcal{M}'_y.\ (\mathit{dom}(\mathcal{M}') \backslash \mathsf{tidset}(\mathfrak{T}'_x), \mathcal{M}'_y) < (\mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x), \mathcal{M}_y) \Longrightarrow P(\mathcal{M}', \mathfrak{T}'_x, \mathcal{M}'_y))\\ \quad \Longrightarrow P(\mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y)\end{array}$$
   The proof is by inversion over $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon))$. We only have two possible cases:
   i. $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}_y}{}^{+}$ (**skip**, _) and $\mathsf{get\_obsv}(\mathfrak{T}_y) = \epsilon$:
   We get $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}_y}{}^{+}$ (**skip**, _) and $\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}_y) = \epsilon$. By the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \epsilon) \in F(S)$.
   ii. $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}'_x}{}^{*} (W'_y, \mathcal{S}'_y)$, $\mathsf{get\_obsv}(\mathfrak{T}'_x) = \epsilon$, $(\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \epsilon))$, $\mathit{dom}(\mathcal{M}_y) = \mathsf{activeThrds}(W_y) \neq \emptyset$ and $\forall \mathsf{t} \in \mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})$:
   We first show
   $$(\mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x), \mathcal{M}'_y) < (\mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x), \mathcal{M}_y)\,. \tag{B.8}$$
   By the operational semantics, we know
   $$\mathit{dom}(\mathcal{M}_y) = \mathsf{activeThrds}(W_y) \subseteq \mathsf{activeThrds}(W) = \mathit{dom}(\mathcal{M}).$$
   Since $\mathsf{tidset}(\mathfrak{T}_x) \subseteq \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x)$, we know $\mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x) \subseteq \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x)$. If $\mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x) = \mathit{dom}(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x)$, for any $\mathsf{t} \in \mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x)$, we know $\mathsf{t} \in \mathit{dom}(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}'_x)$. Thus $\mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})$. Thus we have proved (B.8).
   By the transfinite induction hypothesis, we have

$$\begin{array}{l} \forall W'_y, \mathcal{S}'_y. \\ ((W,\mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}'_x}{}^* (W'_y, \mathcal{S}'_y)) \wedge (\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}'_x) = \epsilon) \\ \wedge\ (\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \epsilon)) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})) \\ \Longrightarrow \\ (W, \mathcal{S}, \epsilon) \in F(S) \end{array}$$

Thus we get $(W, \mathcal{S}, \epsilon) \in F(S)$.

4. $\mathfrak{T}_o = \iota :: \mathfrak{T}_a :: \mathfrak{T}_b$, $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x}{}^+ (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\mathfrak{T}_x) = \iota :: \mathfrak{T}_a$, $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b))$, $dom(\mathcal{M}) = \mathsf{activeThrds}(W)$ and $\forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t})$:
We want to prove $(W, \mathcal{S}, \epsilon) \in F(S)$. We have two cases:
   (a) $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$:
   From $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b))$, we know $(W_y, \mathcal{S}_y, \mathfrak{T}_b) \in S$. From the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
   (b) $dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x) \neq \emptyset$:
   By transfinite induction over the metric $(dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x), \mathcal{M}_y)$.
   $$\text{Transfinite Induction Principle: } (\forall M.\ (\forall M'.\ M' < M \Longrightarrow P(M')) \Longrightarrow P(M)\ ) \ \Longrightarrow \forall M. P(M)$$
   The order over the metrics, $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$, is defined as follows.
   $$(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1) \ \text{ iff } \ (\xi_2 \subset \xi_1) \ \vee \ (\xi_2 = \xi_1 \neq \emptyset) \wedge (\forall \mathsf{t} \in \xi_2.\ \mathcal{M}_2(\mathsf{t}) < \mathcal{M}_1(\mathsf{t}))$$
   It is clear that $(\xi_2, \mathcal{M}_2) < (\xi_1, \mathcal{M}_1)$ is a well-founded order.
   We view our goal as $\forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y. P(\mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y)$, which is reformulated as follows.
   $$\begin{array}{l} \forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y. \\ \quad \forall W_y, \mathcal{S}_y, \iota, \mathfrak{T}_a, \mathfrak{T}_b. \\ \quad (\mathfrak{T}_o = \iota :: \mathfrak{T}_a :: \mathfrak{T}_b) \wedge ((W, \mathcal{S}) \xmapsto{\mathfrak{T}_x}{}^+ (W_y, \mathcal{S}_y)) \wedge (\mathsf{get\_obsv}(\mathfrak{T}_x) = \iota :: \mathfrak{T}_a) \\ \quad \wedge\ (\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b)) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t})) \\ \quad \Longrightarrow \\ \quad (W, \mathcal{S}, \mathfrak{T}_o) \in F(S) \end{array}$$
   By the transfinite induction principle, we only need to prove
   $$\begin{array}{l} \forall \mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y. \\ (\forall \mathcal{M}', \mathfrak{T}'_x, \mathcal{M}'_y.\ (dom(\mathcal{M}') \backslash \mathsf{tidset}(\mathfrak{T}'_x), \mathcal{M}'_y) < (dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathfrak{T}_x), \mathcal{M}_y) \Longrightarrow P(\mathcal{M}', \mathfrak{T}'_x, \mathcal{M}'_y)) \\ \Longrightarrow P(\mathcal{M}, \mathfrak{T}_x, \mathcal{M}_y) \end{array}$$
   The proof is by inversion over $(\mathcal{M}_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b))$. We have four cases:
      i. $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}_y}{}^+$ **abort** and $\mathsf{get\_obsv}(\mathfrak{T}_y) = \mathfrak{T}_b$:
      We get $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}_y}{}^+$ **abort** and $\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}_y) = \mathfrak{T}_o$. By the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
      ii. $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}_y}{}^+ (\textbf{skip}, \_)$ and $\mathsf{get\_obsv}(\mathfrak{T}_y) = \mathfrak{T}_b$:
      We get $(W, \mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}_y}{}^+ (\textbf{skip}, \_)$ and $\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}_y) = \mathfrak{T}_o$. By the definition of F (at the middle part in Figure 24), we know $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
      iii. $\mathfrak{T}_b = \epsilon$, $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}'_x}{}^* (W'_y, \mathcal{S}'_y)$, $\mathsf{get\_obsv}(\mathfrak{T}'_x) = \epsilon$, $(\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \epsilon))$, $dom(\mathcal{M}_y) = \mathsf{activeThrds}(W_y) \neq \emptyset$ and $\forall \mathsf{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})$:
      With (B.8), by the transfinite induction hypothesis, we have
      $$\begin{array}{l} \forall W'_y, \mathcal{S}'_y, \iota, \mathfrak{T}_a, \mathfrak{T}_b. \\ (\mathfrak{T}_o = \iota :: \mathfrak{T}_a :: \mathfrak{T}_b) \wedge ((W, \mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}'_x}{}^+ (W'_y, \mathcal{S}'_y)) \wedge (\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}'_x) = \iota :: \mathfrak{T}_a) \\ \wedge\ (\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \mathfrak{T}_b)) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})) \\ \Longrightarrow \\ (W, \mathcal{S}, \mathfrak{T}_o) \in F(S) \end{array}$$
      Thus we can get $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.
      iv. $\mathfrak{T}_b = \iota' :: \mathfrak{T}'_a :: \mathfrak{T}'_b$, $(W_y, \mathcal{S}_y) \xmapsto{\mathfrak{T}'_x}{}^+ (W'_y, \mathcal{S}'_y)$, $\mathsf{get\_obsv}(\mathfrak{T}'_x) = \iota' :: \mathfrak{T}'_a$, $(\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \mathfrak{T}'_b))$, $dom(\mathcal{M}_y) = \mathsf{activeThrds}(W_y)$ and $\forall \mathsf{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})$:
      With (B.8), by the transfinite induction hypothesis, we have
      $$\begin{array}{l} \forall W'_y, \mathcal{S}'_y, \iota, \mathfrak{T}_a, \iota', \mathfrak{T}'_a, \mathfrak{T}'_b. \\ (\mathfrak{T}_o = \iota :: \mathfrak{T}_a :: \iota' :: \mathfrak{T}'_a :: \mathfrak{T}'_b) \wedge ((W, \mathcal{S}) \xmapsto{\mathfrak{T}_x :: \mathfrak{T}'_x}{}^+ (W'_y, \mathcal{S}'_y)) \\ \wedge\ (\mathsf{get\_obsv}(\mathfrak{T}_x :: \mathfrak{T}'_x) = \iota :: \mathfrak{T}_a :: \iota' :: \mathfrak{T}'_a) \\ \wedge\ (\mathcal{M}'_y \models \mathcal{O}^{\mathsf{m}}_{f\omega}(W'_y, \mathcal{S}'_y, \mathfrak{T}'_b)) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}_y) \backslash \mathsf{tidset}(\mathfrak{T}_x :: \mathfrak{T}'_x).\ \mathcal{M}'_y(\mathsf{t}) < \mathcal{M}_y(\mathsf{t})) \\ \Longrightarrow \\ (W, \mathcal{S}, \mathfrak{T}_o) \in F(S) \end{array}$$
      Thus we can get $(W, \mathcal{S}, \mathfrak{T}_o) \in F(S)$.

Then we are done. □

Proof of Lemma 30. First we define

$$\mathsf{roundsub}(\mathfrak{T}, \xi, \mathfrak{T}_x) \text{ iff } (|\mathfrak{T}| \neq \omega) \wedge (\mathfrak{T}_x = \mathfrak{T}) \vee \exists \mathfrak{T}'.\ (\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}') \wedge (|\mathfrak{T}_x| \neq \omega) \wedge (\xi \subseteq \mathsf{tidset}(\mathfrak{T}_x))$$

$$\mathsf{minPos}(\mathfrak{T}, \xi_1, \xi_2) \overset{\text{def}}{=} \begin{cases} \mathsf{minPos}(\mathfrak{T}, \xi_2) & \text{if } \xi_1 = \emptyset \\ \mathsf{minPos}(\mathfrak{T}, \xi_1) & \text{if } \xi_1 \neq \emptyset \end{cases}$$

$$\mathsf{minPos}(\mathfrak{T}, \xi) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } \mathfrak{T} = \epsilon \\ 1 & \text{if } \mathfrak{T} = \iota :: \mathfrak{T}', |\mathfrak{T}| \neq \omega \text{ and } \mathsf{tid}(\iota) \in \xi \\ 1 + \mathsf{minPos}(\mathfrak{T}', \xi) & \text{if } \mathfrak{T} = \iota :: \mathfrak{T}', |\mathfrak{T}| \neq \omega \text{ and } \mathsf{tid}(\iota) \notin \xi \end{cases}$$

We prove the following (B.9) by inversion over $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$.

If $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$, then there exists $\mathfrak{T}_x$ such that $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$. (B.9)

Also, by inversion over $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, we know

$$dom(\mathscr{M}) = dom(\zeta) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W}).$$

Choose $\mathfrak{T}_x$ such that $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$. Next we choose $\mathcal{M}$ to be a function such that the following hold:

(1) $dom(\mathcal{M}) = \mathsf{activeThrds}(W)$.
(2) For any $\mathsf{t} \in dom(\mathcal{M})$, we have $\mathcal{M}(\mathsf{t}) = (\mathscr{M}(\mathsf{t}), (\zeta(\mathsf{t}), dom(\zeta) \backslash \{\mathsf{t}\}), \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}), \{\mathsf{t}\}))$.

We define the order $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$ as a dictionary order:

$$\begin{array}{l} (M', (\xi', \xi'_D), k') < (M, (\xi, \xi_D), k) \text{ iff} \\ \quad (M' < M) \vee (M' = M) \wedge ((\xi', \xi'_D) < (\xi, \xi_D)) \vee (M' = M) \wedge ((\xi', \xi'_D) = (\xi, \xi_D)) \wedge (k' < k) \\ (\xi', \xi'_D) < (\xi, \xi_D) \text{ iff} \\ \quad (\xi' \supset \xi) \wedge (\xi' \subseteq \xi'_D) \wedge (\xi \subseteq \xi_D) \wedge (\xi'_D \subseteq \xi_D) \\ (\xi', \xi'_D) = (\xi, \xi_D) \text{ iff} \\ \quad (\xi' = \xi) \wedge (\xi' \subseteq \xi'_D) \wedge (\xi \subseteq \xi_D) \wedge (\xi'_D \subseteq \xi_D) \end{array}$$

Clearly that $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$ is a well-founded order.

Next we prove: for any $\mathfrak{T}$, W, $\mathcal{S}$, $\mathfrak{T}_o$, $\mathbb{W}$, $\mathbb{S}$, $\mathscr{M}$, ξ, $\mathcal{M}$ and $\mathfrak{T}_x$, if

(1) $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$;
(2) $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$;
(3) $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$;
$dom(\mathcal{M}) = \mathsf{activeThrds}(W)$; and
for any $\mathsf{t} \in dom(\mathcal{M})$, we have $\mathcal{M}(\mathsf{t}) = (\mathscr{M}(\mathsf{t}), (\zeta(\mathsf{t}), dom(\zeta) \backslash \{\mathsf{t}\}), \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}), \{\mathsf{t}\}))$,

then $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$.

By co-induction. We need to prove the following.

(1) $dom(\mathcal{M}) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, we know $dom(\mathscr{M}) = dom(\zeta) = \mathsf{activeThrds}(W) = \mathsf{activeThrds}(\mathbb{W})$.
(2) If $(W, \mathcal{S}) \longmapsto (\mathbf{skip}, \mathcal{S}')$ and $\mathfrak{T} = \epsilon$, then
there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^+ (\mathbf{skip}, \mathbb{S}')$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, we know there exist $\mathbb{T}$ and $\mathbb{S}'$ such that $\mathsf{get\_obsv}(\mathbb{T}) = \epsilon$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^+ (\mathbf{skip}, \mathbb{S}')$.
(3) If $(W, \mathcal{S}) \overset{\iota}{\longmapsto} \mathbf{abort}$ and $\mathfrak{T} = \iota :: \epsilon$, then
there exist t and $\mathbb{T}$ such that $\lfloor \iota \rfloor = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$, $\lfloor \iota \rfloor = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^+ \mathbf{abort}$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, we know there exist t and $\mathbb{T}$ such that $\lfloor \iota \rfloor = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$, $\lfloor \iota \rfloor = \mathsf{get\_obsv}(\mathbb{T})$ and $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^+ \mathbf{abort}$.
(4) If $(W, \mathcal{S}) \overset{\iota}{\longmapsto} (W', \mathcal{S}')$ and $\mathfrak{T} = \iota :: \mathfrak{T}'$, then
there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$ and $\mathcal{M}'$ such that all the following hold:
(a) $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^* (\mathbb{W}', \mathbb{S}')$;
(b) $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, $\mathsf{get\_obsv}(\lfloor \iota \rfloor) = \mathsf{get\_obsv}(\mathbb{T})$, $(\lfloor \iota \rfloor = (\mathsf{t}, \mathbf{term})) \Rightarrow (\lfloor \iota \rfloor = \mathsf{last}(\mathbb{T}))$;
(c) $\mathfrak{T}' \models (W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$;
(d) for any $\mathsf{t}' \in dom(\mathcal{M})$, either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.
Proof: From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$, we know there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$, $\mathscr{M}'$ and ζ' such that all the following hold:
(A) $(\mathbb{W}, \mathbb{S}) \overset{\mathbb{T}}{\longmapsto}{}^* (\mathbb{W}', \mathbb{S}')$;
(B) $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, $\mathsf{get\_obsv}(\lfloor \iota \rfloor) = \mathsf{get\_obsv}(\mathbb{T})$, $(\lfloor \iota \rfloor = (\mathsf{t}, \mathbf{term})) \Rightarrow (\lfloor \iota \rfloor = \mathsf{last}(\mathbb{T}))$;
(C) $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')$;

(D) either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$,
or $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$,
or $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\zeta(\mathsf{t}) \neq \emptyset$ and $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$;

(E) for any $\mathsf{t}' \in dom(\mathscr{M})\backslash\{\mathsf{t}\}$, we have:
either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$,
or $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$,
or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$.

Since $\mathfrak{T} \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)$, by Lemmas 32 and 33, we know there exists $\mathfrak{T}'_o$ such that $\mathfrak{T}_o = (\mathsf{get\_obsv}(\iota)) :: \mathfrak{T}'_o$ and $\mathfrak{T}' \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W', \mathcal{S}', \mathfrak{T}'_o)$.
By (B.9), we know there exists $\mathfrak{T}'_x$ such that $\mathsf{roundsub}(\mathfrak{T}', \mathsf{activeThrds}(W'), \mathfrak{T}'_x)$.
Choose $\mathcal{M}'$ such that $dom(\mathcal{M}') = \mathsf{activeThrds}(W')$ and for any $\mathsf{t} \in dom(\mathcal{M}')$, we have
$\mathcal{M}'(\mathsf{t}) = (\mathscr{M}'(\mathsf{t}), (\zeta'(\mathsf{t}), dom(\zeta')\backslash\{\mathsf{t}\}), \mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}), \{\mathsf{t}\}))$.
By the co-induction hypothesis, we know $\mathfrak{T}' \models (W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$.
Below we prove: either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$. From (D), we know either $\mathsf{t} \in \mathsf{tidset}(\mathbb{T})$, or $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$, or $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\zeta(\mathsf{t}) \neq \emptyset$ and $\zeta(\mathsf{t}) \subseteq \zeta'(\mathsf{t})$.

- If $\mathscr{M}'(\mathsf{t}) < \mathscr{M}(\mathsf{t})$, then $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.
- If $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\emptyset \subset \zeta(\mathsf{t}) \subset \zeta'(\mathsf{t})$, we prove $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$ as follows. From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$ and $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')$, we know
$$dom(\zeta') = \mathsf{activeThrds}(W'), \;\; dom(\zeta) = \mathsf{activeThrds}(W),$$
$$\zeta'(\mathsf{t}) \subseteq (dom(\zeta')\backslash\{\mathsf{t}\}), \;\; \zeta(\mathsf{t}) \subseteq (dom(\zeta)\backslash\{\mathsf{t}\}).$$
By the operational semantics, we know $\mathsf{activeThrds}(W') \subseteq \mathsf{activeThrds}(W)$. Thus we have
$$(\zeta'(\mathsf{t}), dom(\zeta')\backslash\{\mathsf{t}\}) < (\zeta(\mathsf{t}), dom(\zeta)\backslash\{\mathsf{t}\}).$$
Thus $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.
- If $\mathscr{M}'(\mathsf{t}) = \mathscr{M}(\mathsf{t})$ and $\emptyset \subset \zeta(\mathsf{t}) = \zeta'(\mathsf{t})$, we prove $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$ as follows. First we have $(\zeta'(\mathsf{t}), dom(\zeta')\backslash\{\mathsf{t}\}) = (\zeta(\mathsf{t}), dom(\zeta)\backslash\{\mathsf{t}\})$.
Below we prove $\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}), \{\mathsf{t}\}) < \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}), \{\mathsf{t}\})$. We know
$$\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}), \{\mathsf{t}\}) = \mathsf{minPos}(\mathfrak{T}'_x, \zeta(\mathsf{t})) \text{ and } \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}), \{\mathsf{t}\}) = \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t})).$$
Since $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we know there exists $\mathfrak{T}_z$ such that $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_z$. Since $\mathfrak{T} \neq \epsilon$ and $\mathsf{activeThrds}(W) \neq \emptyset$, we know $\mathfrak{T}_x \neq \epsilon$. Since $\mathfrak{T} = \iota :: \mathfrak{T}'$, we know there exists $\mathfrak{T}_y$ such that $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathfrak{T}' = \mathfrak{T}_y :: \mathfrak{T}_z$. Since $\mathsf{t} \notin \zeta(\mathsf{t})$ and $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, we know
$$\mathsf{minPos}(\mathfrak{T}_y, \zeta(\mathsf{t})) < \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t})).$$
Next we prove $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}), \mathfrak{T}_y)$. From $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we have two cases:
  - If $|\mathfrak{T}| \neq \omega$ and $\mathfrak{T}_x = \mathfrak{T}$, then $\mathfrak{T}_z = \epsilon$. Thus $|\mathfrak{T}'| \neq \omega$ and $\mathfrak{T}_y = \mathfrak{T}'$.
  - If $|\mathfrak{T}_x| \neq \omega$ and $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$, then $|\mathfrak{T}_y| \neq \omega$ and $\zeta(\mathsf{t}) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$. Since $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, we know $\mathsf{tidset}(\mathfrak{T}_x) = \{\mathsf{t}\} \cup \mathsf{tidset}(\mathfrak{T}_y)$. Since $\mathsf{t} \notin \zeta(\mathsf{t})$, we know $\zeta(\mathsf{t}) \subseteq \mathsf{tidset}(\mathfrak{T}_y)$.

  Thus $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}), \mathfrak{T}_y)$. On the other hand, since $\mathsf{roundsub}(\mathfrak{T}', \mathsf{activeThrds}(W'), \mathfrak{T}'_x)$ and $\zeta(\mathsf{t}) = \zeta'(\mathsf{t}) \subseteq \mathsf{activeThrds}(W')$, we know $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}), \mathfrak{T}'_x)$. Then by Lemma 34, we know
$$\mathsf{minPos}(\mathfrak{T}_y, \zeta(\mathsf{t})) = \mathsf{minPos}(\mathfrak{T}'_x, \zeta(\mathsf{t})).$$
Thus $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.

Next we prove: for any $\mathsf{t}' \in dom(\mathcal{M})\backslash\{\mathsf{t}\}$, either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$. From (E), we know either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$, or $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subseteq \zeta'(\mathsf{t}')$.

- If $\mathscr{M}'(\mathsf{t}') < \mathscr{M}(\mathsf{t}')$, then $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.
- If $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\zeta(\mathsf{t}') \subset \zeta'(\mathsf{t}')$, we prove $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$ as follows. From $(W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond (\mathscr{M}, \zeta)$ and $(W', \mathcal{S}') \preceq (\mathbb{W}', \mathbb{S}') \diamond (\mathscr{M}', \zeta')$, we know
$$dom(\zeta') = \mathsf{activeThrds}(W'), \;\; dom(\zeta) = \mathsf{activeThrds}(W),$$
$$\zeta'(\mathsf{t}') \subseteq (dom(\zeta')\backslash\{\mathsf{t}'\}), \;\; \zeta(\mathsf{t}') \subseteq (dom(\zeta)\backslash\{\mathsf{t}'\}).$$
By the operational semantics, we know $\mathsf{activeThrds}(W') \subseteq \mathsf{activeThrds}(W)$. Thus we have
$$(\zeta'(\mathsf{t}'), dom(\zeta')\backslash\{\mathsf{t}'\}) < (\zeta(\mathsf{t}'), dom(\zeta)\backslash\{\mathsf{t}'\}).$$
Thus $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.
- If $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$ and $\emptyset = \zeta(\mathsf{t}') = \zeta'(\mathsf{t}')$, we prove $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$ as follows. First we have $(\zeta'(\mathsf{t}'), dom(\zeta')\backslash\{\mathsf{t}'\}) = (\zeta(\mathsf{t}'), dom(\zeta)\backslash\{\mathsf{t}'\})$.
Below we prove $\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}'), \{\mathsf{t}'\}) < \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}'), \{\mathsf{t}'\})$. We know
$$\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}'), \{\mathsf{t}'\}) = \mathsf{minPos}(\mathfrak{T}'_x, \{\mathsf{t}'\}) \text{ and } \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}'), \{\mathsf{t}'\}) = \mathsf{minPos}(\mathfrak{T}_x, \{\mathsf{t}'\}).$$
Since $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we know there exists $\mathfrak{T}_z$ such that $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_z$. Since $\mathfrak{T} \neq \epsilon$ and $\mathsf{activeThrds}(W) \neq \emptyset$, we know $\mathfrak{T}_x \neq \epsilon$. Since $\mathfrak{T} = \iota :: \mathfrak{T}'$, we know there exists $\mathfrak{T}_y$ such that $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathfrak{T}' = \mathfrak{T}_y :: \mathfrak{T}_z$. Since $\mathsf{t} \neq \mathsf{t}'$ and $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, we know
$$\mathsf{minPos}(\mathfrak{T}_y, \{\mathsf{t}'\}) < \mathsf{minPos}(\mathfrak{T}_x, \{\mathsf{t}'\}).$$
Next we prove $\mathsf{roundsub}(\mathfrak{T}', \{\mathsf{t}'\}, \mathfrak{T}_y)$. From $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we have two cases:
  - If $|\mathfrak{T}| \neq \omega$ and $\mathfrak{T}_x = \mathfrak{T}$, then $\mathfrak{T}_z = \epsilon$. Thus $|\mathfrak{T}'| \neq \omega$ and $\mathfrak{T}_y = \mathfrak{T}'$.
  - If $|\mathfrak{T}_x| \neq \omega$ and $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$, then $|\mathfrak{T}_y| \neq \omega$ and $\{\mathsf{t}'\} \subseteq \mathsf{tidset}(\mathfrak{T}_x)$. Since $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathsf{t} = \mathsf{tid}(\lfloor \iota \rfloor)$, we know $\mathsf{tidset}(\mathfrak{T}_x) = \{\mathsf{t}\} \cup \mathsf{tidset}(\mathfrak{T}_y)$. Since $\mathsf{t} \neq \mathsf{t}'$, we know $\{\mathsf{t}'\} \subseteq \mathsf{tidset}(\mathfrak{T}_y)$.

Thus $\mathsf{roundsub}(\mathfrak{T}', \{\mathsf{t}'\}, \mathfrak{T}_y)$. On the other hand, since $\mathsf{roundsub}(\mathfrak{T}', \mathsf{activeThrds}(W'), \mathfrak{T}'_x)$ and $\{\mathsf{t}'\} \subseteq \mathsf{activeThrds}(W')$, we know $\mathsf{roundsub}(\mathfrak{T}', \{\mathsf{t}'\}, \mathfrak{T}'_x)$. Then by Lemma 34, we know

$$\mathsf{minPos}(\mathfrak{T}_y, \{\mathsf{t}'\}) = \mathsf{minPos}(\mathfrak{T}'_x, \{\mathsf{t}'\}).$$

Thus $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.

- If $\mathscr{M}'(\mathsf{t}') = \mathscr{M}(\mathsf{t}')$, $\emptyset \subset \zeta(\mathsf{t}') = \zeta'(\mathsf{t}')$ and $\mathsf{t} \notin \zeta(\mathsf{t}')$, we prove $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$ as follows. First we have $(\zeta'(\mathsf{t}'), \mathit{dom}(\zeta')\backslash\{\mathsf{t}'\}) = (\zeta(\mathsf{t}'), \mathit{dom}(\zeta)\backslash\{\mathsf{t}'\})$.
  Below we prove $\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}'), \{\mathsf{t}'\}) < \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}'), \{\mathsf{t}'\})$. We know
  $$\mathsf{minPos}(\mathfrak{T}'_x, \zeta'(\mathsf{t}'), \{\mathsf{t}'\}) = \mathsf{minPos}(\mathfrak{T}'_x, \zeta(\mathsf{t}')) \text{ and } \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}'), \{\mathsf{t}'\}) = \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}')).$$
  Since $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we know there exists $\mathfrak{T}_z$ such that $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_z$. Since $\mathfrak{T} \neq \epsilon$ and $\mathsf{activeThrds}(W) \neq \emptyset$, we know $\mathfrak{T}_x \neq \epsilon$. Since $\mathfrak{T} = \iota :: \mathfrak{T}'$, we know there exists $\mathfrak{T}_y$ such that $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathfrak{T}' = \mathfrak{T}_y :: \mathfrak{T}_z$. Since $\mathsf{t} \notin \zeta(\mathsf{t}')$ and $\mathsf{t} = \mathsf{tid}(\lfloor\iota\rfloor)$, we know
  $$\mathsf{minPos}(\mathfrak{T}_y, \zeta(\mathsf{t}')) < \mathsf{minPos}(\mathfrak{T}_x, \zeta(\mathsf{t}')).$$
  Next we prove $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}'), \mathfrak{T}_y)$. From $\mathsf{roundsub}(\mathfrak{T}, \mathsf{activeThrds}(W), \mathfrak{T}_x)$, we have two cases:
  - If $|\mathfrak{T}| \neq \omega$ and $\mathfrak{T}_x = \mathfrak{T}$, then $\mathfrak{T}_z = \epsilon$. Thus $|\mathfrak{T}'| \neq \omega$ and $\mathfrak{T}_y = \mathfrak{T}'$.
  - If $|\mathfrak{T}_x| \neq \omega$ and $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$, then $|\mathfrak{T}_y| \neq \omega$ and $\zeta(\mathsf{t}') \subseteq \mathsf{tidset}(\mathfrak{T}_x)$. Since $\mathfrak{T}_x = \iota :: \mathfrak{T}_y$ and $\mathsf{t} = \mathsf{tid}(\lfloor\iota\rfloor)$, we know $\mathsf{tidset}(\mathfrak{T}_x) = \{\mathsf{t}\} \cup \mathsf{tidset}(\mathfrak{T}_y)$. Since $\mathsf{t} \notin \zeta(\mathsf{t}')$, we know $\zeta(\mathsf{t}') \subseteq \mathsf{tidset}(\mathfrak{T}_y)$.

  Thus $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}'), \mathfrak{T}_y)$. On the other hand, since $\mathsf{roundsub}(\mathfrak{T}', \mathsf{activeThrds}(W'), \mathfrak{T}'_x)$ and $\zeta(\mathsf{t}') = \zeta'(\mathsf{t}') \subseteq \mathsf{activeThrds}(W')$, we know $\mathsf{roundsub}(\mathfrak{T}', \zeta(\mathsf{t}'), \mathfrak{T}'_x)$. Then by Lemma 34, we know
  $$\mathsf{minPos}(\mathfrak{T}_y, \zeta(\mathsf{t}')) = \mathsf{minPos}(\mathfrak{T}'_x, \zeta(\mathsf{t}')).$$
  Thus $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.

Thus we are done. □

Lemma 32. If $(\iota :: \mathfrak{T}_x) \models \mathcal{O}^{\mathsf{co}}_{\mathit{f\omega}}(W, \mathcal{S}, \mathfrak{T}_a)$, then there exists $\mathfrak{T}_b$ such that $\mathfrak{T}_a = (\mathsf{get\_obsv}(\iota)) :: \mathfrak{T}_b$.

Lemma 33. If $(\iota :: \mathfrak{T}_x) \models \mathcal{O}^{\mathsf{co}}_{\mathit{f\omega}}(W, \mathcal{S}, \mathfrak{T}_a)$, $(W, \mathcal{S}) \overset{\iota}{\longmapsto} (W_x, \mathcal{S}_x)$ and $\mathfrak{T}_a = (\mathsf{get\_obsv}(\iota)) :: \mathfrak{T}_b$, then $\mathfrak{T}_x \models \mathcal{O}^{\mathsf{co}}_{\mathit{f\omega}}(W_x, \mathcal{S}_x, \mathfrak{T}_b)$.

Proof. By co-induction, and then by inversion three times over $(e :: \mathfrak{T}_x) \models \mathcal{O}^{\mathsf{co}}_{\mathit{f\omega}}(W, \mathcal{S}, \mathfrak{T}_a)$. □

Lemma 34. If $\xi \neq \emptyset$, $\mathsf{roundsub}(\mathfrak{T}, \xi, \mathfrak{T}_x)$ and $\mathsf{roundsub}(\mathfrak{T}, \xi, \mathfrak{T}_y)$, then $\mathsf{minPos}(\mathfrak{T}_x, \xi) = \mathsf{minPos}(\mathfrak{T}_y, \xi)$.

Proof. From the premises, we know $|\mathfrak{T}_x| \neq \omega$ and $|\mathfrak{T}_y| \neq \omega$. Suppose $|\mathfrak{T}_x| \leq |\mathfrak{T}_y|$. We have two cases:

- $|\mathfrak{T}| \neq \omega$ and $\mathfrak{T}_x = \mathfrak{T}$:
  We know there exists $\mathfrak{T}'_y$ such that $\mathfrak{T} = \mathfrak{T}_y :: \mathfrak{T}'_y$. Thus $\mathfrak{T}_y = \mathfrak{T}$. Then we have $\mathfrak{T}_x = \mathfrak{T}_y$ and we are done.
- there exists $\mathfrak{T}'_x$ such that $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}'_x$ and $\xi \subseteq \mathsf{tidset}(\mathfrak{T}_x)$:
  We know there exists $\mathfrak{T}'_y$ such that $\mathfrak{T} = \mathfrak{T}_y :: \mathfrak{T}'_y$. Thus there exists $\mathfrak{T}_z$ such that $\mathfrak{T}_y = \mathfrak{T}_x :: \mathfrak{T}_z$ and $\mathfrak{T}'_x = \mathfrak{T}_z :: \mathfrak{T}'_y$. Since $\xi \neq \emptyset$, we know $\mathsf{tidset}(\mathfrak{T}_x) \neq \emptyset$. Thus there exist e and $\mathfrak{T}_0$ such that $\mathfrak{T}_x = e :: \mathfrak{T}_0$. Thus $\mathfrak{T}_y = e :: \mathfrak{T}_0 :: \mathfrak{T}_z$. By induction over $|\mathfrak{T}_0|$.

□

Proof of Lemma 31. By co-induction.

$$\text{Co-induction Principle: } \forall x.\ (\exists S.\ S \subseteq F(S) \wedge x \in S) \implies x \in \text{gfp } F$$

Figure 24 defines F and gfp F (i.e., $\mathcal{O}^{\text{m}}_{f\omega}$ at the bottom part of the figure).

$$S \overset{\text{def}}{=} \{(\mathcal{M}, \mathbb{W}, \mathbb{S}, \mathfrak{T}_o) \mid \exists \mathfrak{T}, W, \mathcal{S}.\ (\mathfrak{T} \models \mathcal{O}^{\text{co}}_{f\omega}(W, \mathcal{S}, \mathfrak{T}_o)) \wedge (\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})\}.$$

So from the co-induction principle, we only need to prove:

$$S \subseteq F(S),\ \text{i.e.,}\ \ \forall \mathcal{M}, \mathbb{W}, \mathbb{S}, \mathfrak{T}_o.\ (\mathcal{M}, \mathbb{W}, \mathbb{S}, \mathfrak{T}_o) \in S \implies (\mathcal{M}, \mathbb{W}, \mathbb{S}, \mathfrak{T}_o) \in F(S)\,.$$

By unfolding S and by inversion over $\mathcal{O}^{\text{co}}_{f\omega}$, we have the following cases.

1. $(\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^n$ **abort** and $\mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o$:
2. $(\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}}{}^n$ (**skip**, _) and $\mathsf{get\_obsv}(\mathfrak{T}) = \mathfrak{T}_o$:
3. $(\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_y$, $\mathfrak{T}_o = \epsilon$, $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}_x}{}^n (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\mathfrak{T}_x) = \epsilon$, $(\mathfrak{T}_y \models \mathcal{O}^{\text{co}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon))$, $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$:
   To prove $(\mathcal{M}, \mathbb{W}, \mathbb{S}, \epsilon) \in F(S)$, we want to prove: there exist $\mathbb{T}_x$, $\mathbb{W}_y$, $\mathbb{S}_y$ and $\mathcal{M}_y$ such that

$$(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}_x}{}^* (\mathbb{W}_y, \mathbb{S}_y),\ \ \mathsf{get\_obsv}(\mathbb{T}_x) = \epsilon,\ \ (\mathcal{M}_y, \mathbb{W}_y, \mathbb{S}_y, \epsilon) \in S,$$
$$dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) \neq \emptyset,\ \ \forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t})\ \ .$$

   Since $(\mathfrak{T}_y \models \mathcal{O}^{\text{co}}_{f\omega}(W_y, \mathcal{S}_y, \epsilon))$, we only need to prove:

$$\begin{array}{l}
\forall n, \mathfrak{T}, W, \mathcal{S}, \mathbb{W}, \mathbb{S}, \mathcal{M}, \mathfrak{T}_x, \mathfrak{T}_y, W_y, \mathcal{S}_y, \mathfrak{T}_o.\\
(n > 0) \wedge (\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}) \wedge (\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_y)\\
\wedge\ ((W, \mathcal{S}) \xrightarrow{\mathfrak{T}_x}{}^n (W_y, \mathcal{S}_y)) \wedge (\mathsf{get\_obsv}(\mathfrak{T}_x) = \mathfrak{T}_o)\\
\implies\\
\exists \mathbb{T}_x, \mathbb{W}_y, \mathbb{S}_y, \mathcal{M}_y.\\
((\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}_x}{}^* (\mathbb{W}_y, \mathbb{S}_y)) \wedge (\mathsf{get\_obsv}(\mathbb{T}_x) = \mathfrak{T}_o)\\
\wedge\ (\mathfrak{T}_y \models (W_y, \mathcal{S}_y) \preceq (\mathbb{W}_y, \mathbb{S}_y) \diamond \mathcal{M}_y)\\
\wedge\ (dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) \neq \emptyset) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t}))
\end{array} \tag{B.10}$$

   By induction over n.
   - *Base case*: $n = 1$. Suppose $\mathfrak{T}_x = e :: \epsilon$. Since $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, we know $dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) = \mathsf{activeThrds}(W) \neq \emptyset$. Also there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$ and $\mathcal{M}'$ such that all the following hold:
     (A) $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}}{}^* (\mathbb{W}', \mathbb{S}')$;
     (B) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \textbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
     (C) $\mathfrak{T}_y \models (W_y, \mathcal{S}_y) \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$;
     (D) for any $\mathsf{t}' \in dom(\mathcal{M})$, either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.
     From (D), we have: $\forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}).\ \mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.
   - *Inductive step*: $n = k + 1$ and $k > 0$. Suppose $\mathfrak{T}_x = e :: \mathfrak{T}'_x$ and

$$(W, \mathcal{S}) \xrightarrow{e} (W_x, \mathcal{S}_x)\ \text{ and }\ (W_x, \mathcal{S}_x) \xrightarrow{\mathfrak{T}'_x}{}^k (W_y, \mathcal{S}_y)\ \ .$$

     Since $\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M}$, we know $dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) = \mathsf{activeThrds}(W) \neq \emptyset$. Also there exist t, $\mathbb{T}$, $\mathbb{W}'$, $\mathbb{S}'$ and $\mathcal{M}'$ such that all the following hold:
     (A) $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}}{}^* (\mathbb{W}', \mathbb{S}')$;
     (B) $\mathsf{t} = \mathsf{tid}(e)$, $\mathsf{get\_obsv}(e) = \mathsf{get\_obsv}(\mathbb{T})$, $(e = (\mathsf{t}, \textbf{term})) \Rightarrow (e = \mathsf{last}(\mathbb{T}))$;
     (C) $(\mathfrak{T}'_x :: \mathfrak{T}_y) \models (W_x, \mathcal{S}_x) \preceq (\mathbb{W}', \mathbb{S}') \diamond \mathcal{M}'$;
     (D) for any $\mathsf{t}' \in dom(\mathcal{M})$, either $\mathsf{t}' \in \mathsf{tidset}(\mathbb{T})$, or $\mathcal{M}'(\mathsf{t}') < \mathcal{M}(\mathsf{t}')$.
     From (D), we have: $\forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}).\ \mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.
     From the induction hypothesis, we have:

$$\begin{array}{l}
\exists \mathbb{T}_x, \mathbb{W}_y, \mathbb{S}_y, \mathcal{M}_y.\\
((\mathbb{W}', \mathbb{S}') \xrightarrow{\mathbb{T}_x}{}^* (\mathbb{W}_y, \mathbb{S}_y)) \wedge (\mathsf{get\_obsv}(\mathbb{T}_x) = \mathsf{get\_obsv}(\mathfrak{T}'_x))\\
\wedge\ (\mathfrak{T}_y \models (W_y, \mathcal{S}_y) \preceq (\mathbb{W}_y, \mathbb{S}_y) \diamond \mathcal{M}_y)\\
\wedge\ (dom(\mathcal{M}') = \mathsf{activeThrds}(\mathbb{W}') \neq \emptyset) \wedge (\forall \mathsf{t} \in dom(\mathcal{M}') \backslash \mathsf{tidset}(\mathbb{T}_x).\ \mathcal{M}_y(\mathsf{t}) < \mathcal{M}'(\mathsf{t}))
\end{array}$$

     Thus we have $(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}::\mathbb{T}_x}{}^* (\mathbb{W}_y, \mathbb{S}_y)$ and $\mathsf{get\_obsv}(\mathbb{T} :: \mathbb{T}_x) = \mathsf{get\_obsv}(e :: \mathfrak{T}'_x) = \mathsf{get\_obsv}(\mathfrak{T}_x)$. Also, for any $\mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T} :: \mathbb{T}_x)$, we know $\mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x)$. Thus $\mathcal{M}'(\mathsf{t}) < \mathcal{M}(\mathsf{t})$. Also, we know $\mathsf{t} \in dom(\mathcal{M}') \backslash \mathsf{tidset}(\mathbb{T}_x)$. Thus $\mathcal{M}_y(\mathsf{t}) < \mathcal{M}'(\mathsf{t})$. By the transitivity of the well-founded order, we have $\mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t})$.
4. $(\mathfrak{T} \models (W, \mathcal{S}) \preceq (\mathbb{W}, \mathbb{S}) \diamond \mathcal{M})$, $n > 0$, $\mathfrak{T} = \mathfrak{T}_x :: \mathfrak{T}_y$, $\mathfrak{T}_o = e_a :: \mathfrak{T}_a :: \mathfrak{T}_b$, $(W, \mathcal{S}) \xrightarrow{\mathfrak{T}_x}{}^n (W_y, \mathcal{S}_y)$, $\mathsf{get\_obsv}(\mathfrak{T}_x) = e_a :: \mathfrak{T}_a$, $(\mathfrak{T}_y \models \mathcal{O}^{\text{co}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b))$, $\mathsf{activeThrds}(W) \subseteq \mathsf{tidset}(\mathfrak{T}_x)$:
   To prove $(\mathcal{M}, \mathbb{W}, \mathbb{S}, \mathfrak{T}_o) \in F(S)$, we want to prove: there exist $\mathbb{T}_x$, $\mathbb{W}_y$, $\mathbb{S}_y$ and $\mathcal{M}_y$ such that

$$(\mathbb{W}, \mathbb{S}) \xrightarrow{\mathbb{T}_x}{}^{*} (\mathbb{W}_y, \mathbb{S}_y), \;\; \mathsf{get\_obsv}(\mathbb{T}_x) = e_a :: \mathfrak{T}_a, \;\; (\mathcal{M}_y, \mathbb{W}_y, \mathbb{S}_y, \mathfrak{T}_b) \in S,$$
$$dom(\mathcal{M}) = \mathsf{activeThrds}(\mathbb{W}) \neq \emptyset, \;\; \forall \mathsf{t} \in dom(\mathcal{M}) \backslash \mathsf{tidset}(\mathbb{T}_x).\, \mathcal{M}_y(\mathsf{t}) < \mathcal{M}(\mathsf{t}) \;\; .$$

Since $(\mathfrak{T}_y \models \mathcal{O}^{\mathsf{co}}_{f\omega}(W_y, \mathcal{S}_y, \mathfrak{T}_b))$, we are done by (B.10).

□

B.5 Equivalence between Contextual Refinement and Starvation-Freedom/Deadlock-Freedom

Below we first define starvation-freedom $\mathsf{starvation\text{-}free}_P(\Pi)$ and deadlock-freedom $\mathsf{deadlock\text{-}free}_P(\Pi)$. They are defined over complete execution traces.

Then we give the full formal definition of linearizability, $\Pi \preceq_P^{\mathsf{lin}} \Gamma$, and present a contextual refinement which observes finite event traces only, $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$. Following earlier work [8, 21, 22], we prove $\Pi \preceq_P^{\mathsf{lin}} \Gamma$ is equivalent to $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$. This equivalence result (Theorem 42) is the basis of our new results that relate $\Pi \sqsubseteq_P \Gamma$ to starvation-freedom and deadlock-freedom of linearizable objects.

Finally, we show the new equivalence results (Theorems 43 and 44) between linearizability, starvation-freedom, deadlock-freedom and the contextual refinement under fair scheduling.

Formalizing starvation-freedom and deadlock-freedom. Following the earlier work [14, 22], we define starvation-freedom and deadlock-freedom over execution traces. We first introduce some auxiliary definitions in Fig. 25. We use $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ to represent the set of *whole* event traces generated by executions starting from $(\lfloor W \rfloor, \mathcal{S})$. Here $(\lfloor W \rfloor, \mathcal{S}) \overset{T}{\longmapsto}{}^* (W', \mathcal{S}')$ represents a zero or multi-step execution of $(\lfloor W \rfloor, \mathcal{S})$ that leads to $(W', \mathcal{S}')$ with the sequence T of events generated; $(\lfloor W \rfloor, \mathcal{S}) \overset{T}{\longmapsto}{}^\omega \cdot$ then represents an infinite execution with the *whole* event trace T, which must be infinite too. We also insert a $(\textbf{spawn}, n)$ event at the head of each event trace to record the number of threads in the program. The lifting $\lfloor W \rfloor$ appends an **end** command at the end of each thread, which generates a $(\mathsf{t}, \textbf{term})$ event to mark the termination of this thread t.

Besides, we say T is fair, i.e., $\mathsf{fair}(T)$, if T is finite or every non-terminating thread t has infinite steps on the trace. Here $T|_\mathsf{t}$ is the subsequence of T consisting of events from thread t only.

Also in Fig. 25, we define $\mathsf{prog\text{-}t}(T)$ to say that every method call in T eventually finishes. Here $\mathsf{pend\_inv}(T(1..i))$ gets the set of pending invocation events in the sub-trace $T(1), \ldots, T(i)$ of T, and $\mathsf{match}(e, T(j))$ says that $T(j)$ is a return event which matches with the invocation e, so the corresponding method call finishes. $\mathsf{prog\text{-}s}(T)$ requires that *some* pending method call finishes. Different from $\mathsf{prog\text{-}t}(T)$, the return event $T(j)$ in $\mathsf{prog\text{-}s}(T)$ does not have to be a matching return of the pending invocation e. We also write $\mathsf{abt}(T)$ to say that T ends with a fault event.

Starvation-freedom and deadlock-freedom require $\mathsf{prog\text{-}t}$ and $\mathsf{prog\text{-}s}$, respectively, in every fair execution. They are defined below.

Definition 35 (Starvation-free objects)**.** $\mathsf{starvation\text{-}free}_P(\Pi)$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \wedge ((\sigma_o, \_) \models P) \wedge \mathsf{fair}(T) \\ \implies \mathsf{prog\text{-}t}(T) \vee \mathsf{abt}(T)\,.\end{array}$$

Here $\circledcirc = \{\mathsf{t}_1 \leadsto \circ, \ldots, \mathsf{t}_n \leadsto \circ\}$.

Definition 36 (Deadlock-free objects)**.** $\mathsf{deadlock\text{-}free}_P(\Pi)$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \wedge ((\sigma_o, \_) \models P) \wedge \mathsf{fair}(T) \\ \implies \mathsf{prog\text{-}s}(T) \vee \mathsf{abt}(T)\,.\end{array}$$

Formalizing linearizability. *Linearizability* describes atomic behaviors of object implementations. Following its standard definition [15], we define linearizability using histories, which are finite event traces T consisting of only method invocations, returns, and object faults. As defined in Fig. 25, we say a return e_2 *matches* an invocation e_1, denoted as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID. An invocation is *pending* in T if no matching return follows it. We use $\mathsf{pend\_inv}(T)$ to get the set of pending invocations in T. We complete a history T by appending zero or more return events, and dropping the remaining pending invocations. The result is denoted by $\mathsf{completions}(T)$. It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 37 (Linearizable histories)**.** $T \preceq^{\mathsf{lin}} T'$ iff both the following hold.

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

That is, T is linearizable with respect to T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an *object* is linearizable iff each of its concurrent histories after *completions* is linearizable with respect to some *legal sequential* history. As defined in Fig. 25, we use $\Gamma \rhd (\Sigma, T')$ to mean that T' is a legal sequential history generated by any client using the specification Γ with an abstract initial state Σ.

Definition 38 (Extensions of a history)**.** $\mathsf{extensions}(T)$ is a set of well-formed histories where we extend T by appending return events. It is inductively defined as follows.

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

Definition 39 (Completions of a history)**.** $\mathsf{truncate}(T)$ is the maximal complete sub-history of T. It is inductively defined by dropping the pending invocations in T.

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

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

$$\mathcal{T}_\omega[\![W,\mathcal{S}]\!] \overset{\text{def}}{=}$$
$$\{(\textbf{spawn}, |W|)::T \mid (\lfloor W \rfloor, \mathcal{S}) \overset{T}{\longmapsto}{}^{\omega} \cdot$$
$$\vee (\lfloor W \rfloor, \mathcal{S}) \overset{T}{\longmapsto}{}^{*} (\textbf{skip}, \_) \vee (\lfloor W \rfloor, \mathcal{S}) \overset{T}{\longmapsto}{}^{*} \textbf{abort}\}$$

$$\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor \overset{\text{def}}{=} \textbf{let}\ \Pi\ \textbf{in}\ (C_1; \textbf{end}) \parallel \ldots \parallel (C_n; \textbf{end})$$

$$|\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n| \overset{\text{def}}{=} n \qquad \mathsf{tnum}((\textbf{spawn}, n)::T) \overset{\text{def}}{=} n$$

$$\mathsf{fair}(T) \text{ iff } |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})$$
$$\mathsf{match}(e_1, e_2) \overset{\text{def}}{=} \mathsf{is\_inv}(e_1) \wedge \mathsf{is\_res}(e_2) \wedge (\mathsf{tid}(e_1) = \mathsf{tid}(e_2))$$
$$\mathsf{pend\_inv}(T) \overset{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg \exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$
$$\mathsf{prog\text{-}t}(T) \text{ iff } \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$
$$\mathsf{prog\text{-}s}(T) \text{ iff } \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$
$$\mathsf{abt}(T) \text{ iff } \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathcal{O}_{f\omega}[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_\omega[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_\omega[\![W,\mathcal{S}]\!]\}$$

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

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

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

Figure 25. Event traces.

Definition 40 (Linearizability of objects)**.** The object implementation Π is linearizable with respect to Γ, written as $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma$, iff

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma, T.\ T \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc)]\!] \wedge ((\sigma, \Sigma) \models P)$$
$$\implies \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Gamma \rhd (\Sigma, T') \wedge T_c \preceq^{\mathsf{lin}} T'$$

Equivalence results.

Definition 41 (Basic contextual refinement)**.** $\Pi \sqsubseteq^{\mathsf{fin}}_{P} \Pi'$ iff

$$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ ((\sigma, \Sigma) \models P) \implies$$
$$\mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc)]\!] \subseteq \mathcal{O}[\![(\textbf{let}\ \Pi'\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \Sigma, \circledcirc)]\!]\ .$$

Theorem 42 (Basic equivalence for linearizability)**.** $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma \iff \Pi \sqsubseteq^{\mathsf{fin}}_{P} \Gamma$.

Theorem 43 (① in Fig. 18)**.** $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma \wedge \mathsf{starvation\text{-}free}_{P}(\Pi) \iff \Pi \sqsubseteq_{P} \Gamma$.

Theorem 44 (② in Fig. 18)**.** $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma \wedge \mathsf{deadlock\text{-}free}_{P}(\Pi) \iff \Pi \sqsubseteq_{\mathsf{wr}_1(P)} \mathsf{wr}_1(\Gamma)$.

In the following subsections, we give the proofs of the above equivalence theorems.

B.5.1 Most general client

The key in our proofs is the use of the Most General Client (MGC). Informally, an MGC is a special client which itself can produce all the possible behaviors produced by any clients. We can define the MGC versions of linearizability, progress properties, and observable refinements, and prove their relationships to the original definitions, which universally quantify over arbitrary client programs. Then we reduce the problems of proving the equivalence between original definitions to proving some relationships between the MGC versions. Since an MGC is a specific client, the latter task is usually much simpler.

In fact, we define three MGCs, which produce different "general" behaviors. We assume $dom(\Pi) = \{f_1, \ldots, f_m\}$, and introduce two instructions to get a random (nondeterministic) value. $x := \textbf{rand}(m)$ assigns x a random integer $i \in [1..m]$, and $x := \textbf{rand}()$ computes an arbitrarily large random integer. Then, for any n, we define MGC_n as follows.

```
$\mathsf{MGT}_{\mathsf{t}}$  $\overset{\text{def}}{=}$  $\textbf{while}\ (\textbf{true})\{$
                 $x_{\mathsf{t}} := \textbf{rand}();\ y_{\mathsf{t}} := \textbf{rand}(m);$
                 $z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});$
             $\}$
$\mathsf{MGC}_n$  $\overset{\text{def}}{=}$  $\parallel_{\mathsf{t} \in [1..n]} \mathsf{MGT}_{\mathsf{t}}$
```

Here x_{t}, y_{t} and z_{t} are all local variables for thread t. Each thread in MGC_n keeps calling a random method with a random argument. We also define MGCp_n, which print out the arguments and return values for method calls.

```
$\mathsf{MGTp}_{\mathsf{t}} \stackrel{\text{def}}{=}$ while (true){
    $x_{\mathsf{t}} := \mathbf{rand}();\ y_{\mathsf{t}} := \mathbf{rand}(m);\ \mathbf{print}(y_{\mathsf{t}}, x_{\mathsf{t}});$
    $z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});\ \mathbf{print}(z_{\mathsf{t}});$
}
$\mathsf{MGCp}_n \stackrel{\text{def}}{=} \|_{\mathsf{t}\in[1..n]}\,\mathsf{MGTp}_{\mathsf{t}}$
```

The third MGC $\mathsf{MGCp1}_n$ is useful to observe the progress of objects. Each thread non-deterministically decides whether to continue calling the methods or to end itself. Also it always prints out 0 before a method call and prints out 1 when the method call finishes.

```
$\mathsf{MGTp1}_{\mathsf{t}} \stackrel{\text{def}}{=}$ while ( $\mathbf{rand}() > 0$ ){
    $x_{\mathsf{t}} := \mathbf{rand}();\ y_{\mathsf{t}} := \mathbf{rand}(m);\ \mathbf{print}(0);$
    $z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}});\ \mathbf{print}(1);$
}
print(2);
$\mathsf{MGCp1}_n \stackrel{\text{def}}{=} \|_{\mathsf{t}\in[1..n]}\,\mathsf{MGTp1}_{\mathsf{t}}$
```

The client memory for the above MGCs should contain the local variables for each thread.

$$\sigma_{\mathsf{MGC}(n)} \stackrel{\text{def}}{=} \{x_{\mathsf{t}} \leadsto \_, y_{\mathsf{t}} \leadsto \_, z_{\mathsf{t}} \leadsto \_ \mid 1 \leq \mathsf{t} \leq n\}$$

B.5.2 Basic equivalence for linearizability

Follow earlier work [22], we first define the MGC versions of "linearizability" and "refinement".

Definition 45. $\Pi \preceq_P^{\mathsf{MGC}} \Gamma$ iff

$$\begin{array}{l}\forall n, \sigma_{\mathsf{MGC}(n)}, \sigma, \Sigma, T.\ T \in \mathcal{H}[\![(\mathbf{let}\,\Pi\,\mathbf{in}\,\mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma, \circledcirc)]\!] \wedge ((\sigma, \Sigma) \models P) \\ \implies \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Gamma \rhd_n (\sigma_{\mathsf{MGC}(n)}, \Sigma, T') \wedge T_c \preceq^{\mathsf{lin}} T'\end{array}$$

where

$$\Gamma \rhd_n (\sigma_{\mathsf{MGC}(n)}, \Sigma, T) \stackrel{\text{def}}{=} T \in \mathcal{H}[\![(\mathbf{let}\,\Gamma\,\mathbf{in}\,\mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \Sigma, \circledcirc)]\!] \wedge \mathsf{seq}(T)\,.$$

Definition 46. $\Pi \subseteqq_P \Gamma$ iff

$$\begin{array}{l}\forall n, \sigma_{\mathsf{MGC}(n)}, \sigma, \Sigma.\ ((\sigma, \Sigma) \models P) \\ \implies \mathcal{H}[\![(\mathbf{let}\,\Pi\,\mathbf{in}\,\mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\,\Gamma\,\mathbf{in}\,\mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \Sigma, \circledcirc)]\!]\,.\end{array}$$

To prove Theorem 42, we prove the following lemmas.

Lemma 47. $\Pi \preceq_P^{\mathsf{lin}} \Gamma \iff \Pi \preceq_P^{\mathsf{MGC}} \Gamma\,.$

Lemma 48. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \iff \Pi \subseteqq_P \Gamma\,.$

Lemma 49. $\Pi \subseteqq_P \Gamma \iff \Pi \preceq_P^{\mathsf{MGC}} \Gamma\,.$

Proof of Lemma 47. We can see $\Pi \preceq_P^{\mathsf{MGC}} \Gamma$ is simply defined by fixing the arbitrarily quantified client programs in $\Pi \preceq_P^{\mathsf{lin}} \Gamma$ (Definition 40) as MGC_n. Intuitively, the key to prove this lemma is to show that any history generated by an arbitrary client program can also be generated by MGC, as shown in the following lemma.

Lemma 50 (MGC is the most general)**.** For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_{\mathsf{MGC}(n)}$ and σ_o, we have $\mathcal{H}[\![(\mathbf{let}\,\Pi\,\mathbf{in}\,C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\,\Pi\,\mathbf{in}\,\mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$.

Proof. We construct a simulation $\lesssim_{\mathsf{MGC}}$ between the client program and the MGC. We need the simulation relation to satisfy the following (B.11).

For any $W_1, \mathcal{S}_1, W_2, \mathcal{S}_2$ and e_1, if $(W_1, \mathcal{S}_1) \lesssim_{\mathsf{MGC}} (W_2, \mathcal{S}_2)$, then
(1) if $(W_1, \mathcal{S}_1) \xmapsto{e_1} \mathbf{abort}$ and $\mathsf{is\_obj\_abt}(e_1)$, then
there exists T_2 such that $(W_2, \mathcal{S}_2) \xmapsto{T_2}{}^* \mathbf{abort}$ and
$e_1 = \mathsf{get\_hist}(T_2)$;
(2) if $(W_1, \mathcal{S}_1) \xmapsto{e_1} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \xmapsto{T_2}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$ and $(W_1', \mathcal{S}_1') \lesssim_{\mathsf{MGC}} (W_2', \mathcal{S}_2')$.
(B.11)

The simulation relation is constructed as shown in Fig. 26(a). That is, for each client thread t, if the left side is in some normal client code, it corresponds to $\mathsf{MGT}_{\mathsf{t}}$ at the right side; otherwise, if the left side is inside a method call, its code is the same as the right side. Informally, the following hold for each thread t.

(1) Each normal client step of the left corresponds to zero step of $\mathsf{MGT}_{\mathsf{t}}$.
(2) Each method invocation corresponds to the steps executing $\mathsf{MGT}_{\mathsf{t}}$ to the same method body, with the same argument.

(3) Each step inside the method body at the left corresponds to the same step at the right.
(4) Each return step at the left corresponds to the same return step (plus a few client steps) executing the right side code to MGT_t.

Then with (B.11), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{H}[\![W_1, \mathcal{S}_1]\!]$.

$$\text{If } (W_1, \mathcal{S}_1) \lesssim_{\mathsf{MGC}} (W_2, \mathcal{S}_2), \text{ then } \mathcal{H}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{H}[\![W_2, \mathcal{S}_2]\!].$$

Also we have

$$(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \lesssim_{\mathsf{MGC}} (\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc))\,,$$

thus we are done. □

Then Lemma 47 is immediate by unfolding the definitions of $\Pi \preceq_P^{\mathsf{MGC}} \Gamma$ and $\Pi \preceq_P^{\mathsf{lin}} \Gamma$, and applying Lemma 50.

Proof of Lemma 48.

1. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \implies \Pi \subseteqq_P \Gamma$:

   To prove this direction, we show that any history generated by MGC_n is "equivalent" to an observable trace generated by MGCp_n, i.e., the following lemma holds.

   Lemma 51. For any n, σ_o and $\sigma_{\mathsf{MGC}(n)}$, both the following holds.

   (a) For any T_1, if $T_1 \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, then
   there exists T_2 such that $T_2 \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$ and $T_1 \approx T_2$.

   (b) For any T_2, if $T_2 \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then
   there exists T_1 such that $T_1 \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$ and $T_1 \approx T_2$.

   Here $T_1 \approx T_2$ is inductively defined as follows.

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{e_1 \approx e_2 \qquad T_1 \approx T_2}{e_1 :: T_1 \approx e_2 :: T_2}$$

$$\frac{}{(\mathsf{t}, f_i, n) \approx (\mathsf{t}, \textbf{out}, (i, n))} \qquad \frac{}{(\mathsf{t}, \textbf{ret}, n) \approx (\mathsf{t}, \textbf{out}, n)}$$

$$\frac{}{(\mathsf{t}, \textbf{obj}, \textbf{abort}) \approx (\mathsf{t}, \textbf{obj}, \textbf{abort})}$$

   Proof. By constructing simulations between MGC_n and MGCp_n. □

2. $\Pi \subseteqq_P \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$:

   Proof. For any n, $C_1, \ldots, C_n$, σ_c, $\sigma_{\mathsf{MGC}(n)}$ and σ_o, by Lemma 50, we know

$$\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]\,.$$

   Since $\Pi \subseteqq_P \Gamma$, we know for any σ_a such that $(\sigma_o, \sigma_a) \models P$, we have

$$\mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,.$$

   Thus, for any T such that

$$T \in \mathcal{T}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,,$$

   there exists T_{MGC} such that $\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T_{\mathsf{MGC}})$ and

$$T_{\mathsf{MGC}} \in \mathcal{T}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,.$$

   Intuitively, we can then replace the object events in T with those in T_{MGC} and keep other events (and the order between them) unchanged. Thus the resulting trace T' satisfies $\mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T)$. We prove T' can be produced by the corresponding abstract client program, that is

$$T' \in \mathcal{T}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.$$

   Then we are done.

   Alternatively, we can actually construct a "simulation" relation $\lesssim$ (see Fig. 26(b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.12).

   For any $W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3$ and e_1,
   if $(W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, then

$$
\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\
\precsim_{\mathsf{MGC}}\ (\textbf{let}\ \Pi\ \textbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n', (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \{1 \rightsquigarrow \kappa_1', \ldots, n \rightsquigarrow \kappa_n'\})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \forall i.\ (C_i, \kappa_i) \precsim^{i}_{\mathsf{MGC}} (C_i', \kappa_i')
\end{array}
$$

$$
(C, \circ) \precsim^{\mathsf{t}}_{\mathsf{MGC}} (\mathsf{MGT_t}, \circ) \qquad (C, (\sigma_l, x, C')) \precsim^{\mathsf{t}}_{\mathsf{MGC}} (C, (\sigma_l, z_{\mathsf{t}}, (\textbf{skip}; \mathsf{MGT_t})))
$$

(a) the program is simulated by MGC

$$
\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\
\precsim (\textbf{let}\ \Gamma\ \textbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n', (\sigma_{\mathsf{MGC}(n)}, \sigma_o', \{1 \rightsquigarrow \kappa_1', \ldots, n \rightsquigarrow \kappa_n'\}); \\
\quad \textbf{let}\ \Gamma\ \textbf{in}\ C_1'' \,\|\, \ldots \,\|\, C_n'', (\sigma_c, \sigma_o', \{1 \rightsquigarrow \kappa_1'', \ldots, n \rightsquigarrow \kappa_n''\})) \\
\text{where } \forall i.\ (C_i, \kappa_i) \precsim (C_i', \kappa_i'; C_i'', \kappa_i'') \\
\text{and } \mathcal{H}[\![\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})]\!] \\
\quad \subseteq \mathcal{H}[\![\textbf{let}\ \Gamma\ \textbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n', (\sigma_{\mathsf{MGC}(n)}, \sigma_o', \{1 \rightsquigarrow \kappa_1', \ldots, n \rightsquigarrow \kappa_n'\})]\!]
\end{array}
$$

$$
(C, \circ) \precsim (C', \circ; C, \circ) \qquad (C, (\sigma_l, x, C_c)) \precsim (C', (\sigma_l', x', C_c'); C', (\sigma_l', x, C_c))
$$

(b) the concrete program is "simulated" by the abstract MGC and the abstract program

Figure 26. Simulations between programs and MGC.

(1) if $(W_1, \mathcal{S}_1) \xrightarrow{e_1} \textbf{abort}$, then there exists T_3 such that
$(W_3, \mathcal{S}_3) \xrightarrow{T_3}{}^* \textbf{abort}$ and $e_1 = \mathsf{get\_obsv}(T_3)$;

(2) if $(W_1, \mathcal{S}_1) \xrightarrow{e_1} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_clt}(e_1)$, then
there exist W_3' and $\mathcal{S}_3'$ such that $(W_3, \mathcal{S}_3) \xrightarrow{e_1} (W_3', \mathcal{S}_3')$ and $(W_1', \mathcal{S}_1') \precsim (W_2, \mathcal{S}_2; W_3', \mathcal{S}_3')$.

(3) if $(W_1, \mathcal{S}_1) \xrightarrow{e_1} (W_1', \mathcal{S}_1')$ and $\mathsf{is\_obj}(e_1)$, then
there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \xrightarrow{T_2}{}^* (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \xrightarrow{T_3}{}^* (W_3', \mathcal{S}_3')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$, $\mathsf{get\_obj}(T_2) = T_3$ and
$(W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.

(B.12)

That is, the executions of the abstract program W_3 follow the concrete program W_1 for client steps and the abstract MGC W_2 for object steps. Here we use $\mathsf{get\_obj}(T)$ to get the sub-trace of T consisting of object events only. We can prove the following from (B.12).

$$
\text{If } (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3), \text{ then } \mathcal{O}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{O}[\![W_3, \mathcal{S}_3]\!].
$$

Initially,

$$
\begin{array}{l}
(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\
\precsim (\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc))
\end{array}
$$

holds, and we are done. □

Proof of Lemma 49.

1. $\Pi \sqsubseteq_P \Gamma \implies \Pi \preceq_P^{\mathsf{MGC}} \Gamma$:

   The premise $\Pi \sqsubseteq_P \Gamma$ tells us that every history generated by using the concrete object Π can also be generated with the abstract object Γ. Thus, to prove the concrete object Π is linearizable, we only need to show the abstract object Γ (whose methods are atomic) is linearizable with respect to itself.

   Lemma 52 (Γ is linearizable)**.**
   For any n, $\sigma_{\mathsf{MGC}(n)}$, σ_a and T, if $T \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then there exist T_c and T' such that $T_c \in \mathsf{completions}(T)$, $T_c \preceq^{\mathsf{lin}} T'$, $\mathsf{seq}(T')$ and $T' \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$.

   Proof. From the execution that generates T, we construct another execution as follows. We postpone every invocation step and advance the latter return step to the single step of the atomic method body in between. We prove the resulting execution generates a history T' satisfying all the requirements. □

2. $\Pi \preceq_P^{\mathsf{MGC}} \Gamma \implies \Pi \sqsubseteq_P \Gamma$:

   The key is to prove that every linearizable history can be generated by the MGC with the *abstract* object Γ.

   Lemma 53 (Rearrangement)**.**
   For any n, $\sigma_{\mathsf{MGC}(n)}$, σ_a, T and T', if $T \preceq^{\mathsf{lin}} T'$, $\mathsf{seq}(T')$ and $T' \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$, then
   $T \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]$.

Proof. We construct the execution generating T, where the order to execute the atomic method body for concurrent method calls simply follows the order of the pairs of invocation and subsequent return events in T'. The detailed proof is by induction over the length of T. $\Box$

B.5.3 Equivalence for starvation-freedom

By Theorem 42, the goal is reduced to the following:

$$\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{starvation\text{-}free}_P(\Pi) \iff \Pi \sqsubseteq_P \Gamma\,.$$

We first define the MGC version of starvation-freedom.

Definition 54. $\mathsf{SF\text{-}MGC}_P(\Pi)$, iff

$$\begin{array}{l}\forall n, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \wedge \mathsf{fair}(T) \wedge ((\sigma_o, \_) \models P) \\ \implies \mathsf{prog\text{-}t}(T) \vee \mathsf{abt}(T)\,.\end{array}$$

We only need to prove the following lemmas.

Lemma 55. $\Pi \sqsubseteq_P \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma\,.$

Lemma 56. $\Pi \sqsubseteq_P \Gamma \implies \mathsf{SF\text{-}MGC}_P(\Pi)\,.$

Lemma 57. $\mathsf{SF\text{-}MGC}_P(\Pi) \implies \mathsf{starvation\text{-}free}_P(\Pi)\,.$

Lemma 58. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{starvation\text{-}free}_P(\Pi) \implies \Pi \sqsubseteq_P \Gamma\,.$

Proof of Lemma 55. For any n, $C_1, \ldots, C_n$, σ_c, σ_o and σ_a such that $(\sigma_o, \sigma_a) \models P$, for any T, if

$$T \in \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

we know there exists T_1 such that $T = \mathsf{get\_obsv}(T_1)$ and

$$T_1 \in \mathcal{T}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

We can prove: there exists T_1' and T_1'' such that

$$T_1'' = (\textbf{spawn}, n) :: T_1 :: T_1' \ , \ \ \mathsf{fair}(T_1'') \ \text{ and } \ T_1'' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!].$$

Since $\Pi \sqsubseteq_P \Gamma$, we know there exists T_2'' such that

$$T_2'' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] \ , \ \ \mathsf{fair}(T_2'') \ \text{ and } \ \mathsf{get\_obsv}(T_2'') = \mathsf{get\_obsv}(T_1'') = T :: \mathsf{get\_obsv}(T_1')\,.$$

Thus there exists T_2 such that

$$T_2 \in \mathcal{T}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] \ \text{ and } \ \mathsf{get\_obsv}(T_2) = T.$$

Thus $T \in \mathcal{O}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]$ and we are done.

Proof of Lemma 56. For any n, σ_o, σ_a and T such that $T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $\mathsf{fair}(T)$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg\mathsf{abt}(T)\,,$$

then by the operational semantics, we only need to prove:

for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then there exists $j > i$ such that $\mathsf{match}(e, T(j))$.

Suppose it does not hold. Then we know there exists t_0 such that

$$\exists i.\ \forall j.\ j \geq i \Rightarrow (T|_{\mathsf{t}_0})(j) = (\mathsf{t}_0, \textbf{obj})\,.$$

Thus we have

$$|(\mathsf{get\_obsv}(T)|_{\mathsf{t}_0})| < i \ \text{ and } \ \mathsf{last}(\mathsf{get\_obsv}(T)|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 0)\ .$$

Since $\Pi \sqsubseteq_P \Gamma$, we know:

$$\mathcal{O}_{f\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_{f\omega}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,.$$

Thus there exists T' such that

$$T' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,, \ \ \mathsf{fair}(T') \ \text{ and } \ \mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T)\,.$$

Thus $\mathsf{last}(\mathsf{get\_obsv}(T')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 0)$. But this is impossible from the operational semantics, $\mathsf{fair}(T')$ and the fact that Γ is atomic and total. Thus we are done.

Proof of Lemma 57. Similar to the proof of Lemma 47, we first prove the following two lemmas (Lemma 59 and Lemma 60). Then, from $\mathsf{SF\text{-}MGC}_P(\Pi)$, we know: if $T' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and $\mathsf{fair}(T')$, then either $\mathsf{prog\text{-}t}(T')$ or $\mathsf{abt}(T')$. By the definition of $\mathsf{MGCp1}_n$ and the operational semantics, we know either $\mathsf{prog\text{-}t}(T')$ or $\mathsf{obj\_abt}(T')$. From Lemmas 59 and 60, we know: if $T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and $\mathsf{fair}(T)$, then either $\mathsf{clt\_abt}(T)$, or $\mathsf{prog\text{-}t}(T)$, or $\mathsf{obj\_abt}(T)$. Thus we have $\mathsf{starvation\text{-}free}_P(\Pi)$.

Lemma 59 (MGC is the most general)**.** For any n, $C_1, \ldots, C_n$, σ_c, σ_o and T, if

$$T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \text{ and } \mathsf{fair}(T),$$

then one of the following holds:

(1) $\mathsf{clt\_abt}(T)$; or
(2) there exists T' such that

$$(T' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]) \wedge (\mathsf{get\_obj}(T) = \mathsf{get\_obj}(T')) \wedge \mathsf{fair}(T').$$

Here $\mathsf{get\_obj}(T)$ gets the sub-trace consisting of only object events (i.e., method invocation, return, object fault and normal object events).

Proof. We construct a simulation relation between the arbitrary client program and the MGC, under the fixed scheduling T for the client at the left side. □

Lemma 60. For any T and T', if $\mathsf{get\_obj}(T) = \mathsf{get\_obj}(T')$ and $\mathsf{prog\text{-}t}(T')$, then $\mathsf{prog\text{-}t}(T)$.

Proof. By unfolding the definitions. □

Proof of Lemma 58. The key is to show the following (B.13).

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a such that $(\sigma_o, \sigma_a) \models P$,
if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, then there exists T_a such that
$(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$, $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and $\mathsf{fair}(T_a)$.
(B.13)

Its proof is similar to the proof of $\Pi \sqsubseteq_P \Gamma \Longrightarrow \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$. We can replace the object events in the concrete *infinite* trace T with those generated by MGC using Γ. The resulting trace T_a satisfies $\mathsf{get\_obsv}(T_a) = \mathsf{get\_obsv}(T)$. From $\mathsf{starvation\text{-}free}_P(\Pi)$, we can show T_a must be infinite and fair.

In detail, we construct a "simulation" relation $\lesssim$ (see Fig. 26(b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.14). (It is derived from (B.12).) Here we use $T \backslash (\_, \textbf{obj})$ for a sub-trace resulting from removing all the events of the form $(\_, \textbf{obj})$ in T.

For any $W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3$ and e_1,
if $(W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$ and $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$,
then there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^{*} (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{*} (W_3', \mathcal{S}_3')$,
$T_3 \backslash (\_, \textbf{obj}) = e_1 \backslash (\_, \textbf{obj})$ and $(W_1', \mathcal{S}_1') \lesssim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.
(B.14)

With (B.14), we can prove the following (B.15) by induction over the length of T_1.

For any $W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3$ and T_1,
if $(W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^{+} (W_1', \mathcal{S}_1')$ and $\mathsf{last}(T_1) \neq (\_, \textbf{obj})$,
then there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^{*} (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{+} (W_3', \mathcal{S}_3')$,
$T_1 \backslash (\_, \textbf{obj}) = T_3 \backslash (\_, \textbf{obj})$ and $(W_1', \mathcal{S}_1') \lesssim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.
(B.15)

Then we prove the following (B.16) by co-induction.

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o, W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3, T_0$ and T_1,
if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T_0}{\longmapsto}{}^{*} (W_1, \mathcal{S}_1)$,
$(W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{prog\text{-}t}(T_0 :: T_1)$,
then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{\omega} \cdot$ and $T_1 \backslash (\_, \textbf{obj}) = T_3 \backslash (\_, \textbf{obj})$.
(B.16)

Also we can prove: for any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a, if $(\sigma_o, \sigma_a) \models P$, then

$$\begin{array}{l}(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \\ \lesssim (\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)).\end{array}$$

If $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, by $\mathsf{starvation\text{-}free}_P(\Pi)$, we know $\mathsf{prog\text{-}t}(T)$. Then from (B.16) we get: there exists T_a such that

$$(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot, \text{ and } T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj}).$$

Thus we know $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

Below we prove $\mathsf{fair}(T_a)$. Since $\mathsf{fair}(T)$ and $|T| = \omega$, we know for any t,

$$\text{either } |(T|_{\mathsf{t}})| = \omega, \text{ or } \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term}).$$

(a) $\mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})$:

Since $T\backslash(\_, \textbf{obj}) = T_a\backslash(\_, \textbf{obj})$ and by the operational semantics, we know $\mathsf{last}(T_a|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term})$.

(b) $|(T|_{\mathsf{t}})| = \omega$:

Since $T\backslash(\_, \textbf{obj}) = T_a\backslash(\_, \textbf{obj})$, we know

$$(T|_{\mathsf{t}})\backslash(\mathsf{t}, \textbf{obj}) = (T_a|_{\mathsf{t}})\backslash(\mathsf{t}, \textbf{obj}) .$$

Suppose $|(T_a|_{\mathsf{t}})| \neq \omega$. Then we know $|\,((T_a|_{\mathsf{t}})\backslash(\mathsf{t}, \textbf{obj}))\,| \neq \omega$. Thus

$$\exists i.\, \forall j.\, j \geq i \;\Rightarrow\; (T|_{\mathsf{t}})(j) = (\mathsf{t}, \textbf{obj}) .$$

By the operational semantics, we know there exists i such that

$$\mathsf{tid}(T(i)) = \mathsf{t}\,, \quad \mathsf{is\_inv}(T(i))\,, \quad \text{and} \quad \forall j.\, j \geq i \;\Rightarrow\; \neg\mathsf{match}(T(i), T(j)) .$$

This contradicts the premise $\mathsf{prog\text{-}t}(T)$. Thus we know $|(T_a|_{\mathsf{t}})| = \omega$.

Thus $\mathsf{fair}(T_a)$ holds and we are done.

B.5.4 Equivalence for deadlock-freedom

We first define the following contextual refinement $\widehat{\sqsubseteq}$ which assumes fair scheduling at the concrete side only.

Definition 61. $\Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma$ iff

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\; ((\sigma, \Sigma) \models P) \Longrightarrow \\ \mathcal{O}_{f\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma, \circledcirc)]\!] \subseteq \mathcal{O}_{\omega}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \Sigma, \circledcirc)]\!] .\end{array}$$

We also define the assertion *id* for the identity relation between the lower-level and the higher-level states:

$$(\sigma, \Sigma) \models \mathit{id} \;\text{ iff }\; \sigma = \Sigma$$

Then $\mathsf{wr}_{\mathtt{l}}(\mathit{id})$ adds the shared variable $\mathtt{l}$ to the higher-level states, and $\mathsf{wr}_{\mathtt{l}}^{-1}(\mathit{id})$ removes $\mathtt{l}$ which should be 0:

$$\begin{array}{lll}(\sigma, \Sigma) \models \mathsf{wr}_{\mathtt{l}}(\mathit{id}) & \text{iff} & \Sigma = \sigma \uplus \{\mathtt{l} \leadsto 0\} \\ (\sigma, \Sigma) \models \mathsf{wr}_{\mathtt{l}}^{-1}(\mathit{id}) & \text{iff} & \sigma = \Sigma \uplus \{\mathtt{l} \leadsto 0\}\end{array}$$

To prove Theorem 44, we prove the following:

$$\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \iff \Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma \tag{B.17}$$

$$\Pi \sqsubseteq_{\mathsf{wr}_{\mathtt{l}}(P)} \mathsf{wr}_{\mathtt{l}}(\Gamma) \implies \Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma \tag{B.18}$$

$$\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \implies \Pi \sqsubseteq_{\mathsf{wr}_{\mathtt{l}}(P)} \mathsf{wr}_{\mathtt{l}}(\Gamma) \tag{B.19}$$

Proofs of (B.17). We first define the MGC version of deadlock-freedom.

Definition 62. $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$, iff

$$\begin{array}{l}\forall n, \sigma_o, T.\; T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \wedge \mathsf{fair}(T) \wedge ((\sigma_o, \_) \models P) \\ \Longrightarrow \mathsf{prog\text{-}s}(T) \vee \mathsf{abt}(T) .\end{array}$$

We only need to prove the following lemmas.

Lemma 63. $\Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma \implies \Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma$.

Lemma 64. $\Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma \implies \mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$.

Lemma 65. $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi) \implies \mathsf{deadlock\text{-}free}_P(\Pi)$.

Lemma 66. $\Pi \sqsubseteq_P^{\mathsf{fin}} \Gamma \wedge \mathsf{deadlock\text{-}free}_P(\Pi) \implies \Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma$.

Proof of Lemma 63. Similar to the proof of Lemma 55. □

Proof of Lemma 64. Similar to the proof of Lemma 56. For any n, σ_o, σ_a and T such that $T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $\mathsf{fair}(T)$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg\mathsf{abt}(T)\,,$$

then by the operational semantics, we only need to prove:

for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$ holds.

Suppose it does not hold. Then we know:

$$\exists i.\ \forall j.\ j \geq i \Rightarrow T(j) = (\_, \textbf{obj})\,.$$

Since $\mathsf{fair}(T)$, we have

$$\forall \mathsf{t}_0 \in [1..n].\ \mathsf{last}(\mathsf{get\_obsv}(T)|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 0) \vee \mathsf{last}(\mathsf{get\_obsv}(T)|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 2)\ .$$

Since $\Pi \mathrel{\widehat{\sqsubseteq}_P} \Gamma$, we know:

$$\mathcal{O}_{f\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_{\omega}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!]\,.$$

Thus there exists T' such that

$$T' \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc)]\!] \quad \text{and} \quad \mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T)\,.$$

Thus $\forall \mathsf{t}_0 \in [1..n].\ \mathsf{last}(\mathsf{get\_obsv}(T')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 0) \vee \mathsf{last}(\mathsf{get\_obsv}(T')|_{\mathsf{t}_0}) = (\mathsf{t}_0, \textbf{out}, 2)$. But this is impossible from the operational semantics and the fact that Γ is atomic and total. Thus we are done. □

Proof of Lemma 65. Similar to the proof of Lemma 57. From $\mathsf{deadlock\text{-}free}_P^{\mathsf{MGC}}(\Pi)$, we get: if $T' \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\sigma_{\mathsf{MGC}(n)}, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and $\mathsf{fair}(T')$, then either $\mathsf{prog\text{-}s}(T')$ or $\mathsf{abt}(T')$. By the definition of $\mathsf{MGCp1}_n$ and the operational semantics, we know either $\mathsf{prog\text{-}s}(T')$ or $\mathsf{obj\_abt}(T')$. From Lemma 59, we can prove: if $T \in \mathcal{T}_{\omega}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, $(\sigma_o, \_) \models P$ and $\mathsf{fair}(T)$, then either $\mathsf{clt\_abt}(T)$, or $\mathsf{prog\text{-}s}(T)$, or $\mathsf{obj\_abt}(T)$. Thus we have $\mathsf{deadlock\text{-}free}_P(\Pi)$. □

Proofs of Lemma 66. Similar to the proof of Lemma 58. The key is to show the following (B.20).

> For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a such that $(\sigma_o, \sigma_a) \models P$,
> if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, then there exists T_a such that
> $(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{T_a}{}^{\omega} \cdot$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.
> (B.20)

As in the proofs for (B.13), we construct a "simulation" relation $\lesssim$ (see Fig. 26(b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.21).

> For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o, W_1, \mathcal{S}_1, W_2, \mathcal{S}_2, W_3, \mathcal{S}_3, T_0$ and T_1,
> if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_0}{}^{*} (W_1, \mathcal{S}_1)$,
> $(W_1, \mathcal{S}_1) \lesssim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \xmapsto{T_1}{}^{\omega} \cdot$ and $\mathsf{prog\text{-}s}(T_0 :: T_1)$,
> then there exists T_3 such that $(W_3, \mathcal{S}_3) \xmapsto{T_3}{}^{\omega} \cdot$ and $T_1 \backslash (\_, \textbf{obj}) = T_3 \backslash (\_, \textbf{obj})$.
> (B.21)

Also we can prove: for any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_a, if $(\sigma_o, \sigma_a) \models P$, then

$$\begin{array}{l}(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \\ \lesssim (\textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{MGC}_n, (\sigma_{\mathsf{MGC}(n)}, \sigma_a, \circledcirc);\ \lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc))\,.\end{array}$$

If $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, by $\mathsf{deadlock\text{-}free}_P(\Pi)$, we know $\mathsf{prog\text{-}s}(T)$. Then from (B.21) we get: there exists T_a such that

$$(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{T_a}{}^{\omega} \cdot, \text{ and } T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj}).$$

Thus we know $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and we are done. □

Proofs of (B.18). We only need to prove the following:

$$\mathsf{wr}_1(\Gamma) \mathrel{\widehat{\sqsubseteq}_{\mathsf{wr}_1^{-1}(id)}} \Gamma$$

By (B.17), we only need to show:

$$\mathsf{wr}_1(\Gamma) \sqsubseteq^{\mathsf{fin}}_{\mathsf{wr}_1^{-1}(id)} \Gamma \tag{B.22}$$

$$\mathsf{deadlock\text{-}free}_{\mathsf{wr}_1^{-1}(id)}(\mathsf{wr}_1(\Gamma)) \tag{B.23}$$

Proof of (B.22). We want to show: for any $n, C_1, \ldots, C_n, \sigma_c$ and σ_a,

$$\begin{array}{l}\mathcal{O}[\![(\textbf{let}\ \mathsf{wr}_1(\Gamma)\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a \uplus \{\mathtt{l} \leadsto 0\}))]\!] \\ \subseteq \mathcal{O}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!]\,.\end{array}$$

We construct a simulation relation $\lesssim$ as follows.

$$
\begin{array}{l}
(\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C'_1 \,\|\, \ldots \,\|\, C'_n, (\sigma_c, \sigma'_a, \{1 \leadsto \kappa'_1, \ldots, n \leadsto \kappa'_n\})) \\
\lesssim (\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \{1 \leadsto \kappa_1, \ldots, n \leadsto \kappa_n\})) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \forall i.\ (C'_i, \sigma'_a, \kappa'_i) \lesssim_i (C_i, \sigma_a, \kappa_i)
\end{array}
$$

$$
\begin{array}{l}
(C, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \circ) \lesssim_{\mathsf{t}} (C, \sigma_a, \circ) \\
(\mathsf{wr}_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}'_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}''_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \mathsf{t}\}, \kappa') \lesssim_{\mathsf{t}} (\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}'''_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\langle C\rangle; \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\langle C\rangle; \mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}'_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}''_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \mathsf{t}\}, \kappa') \lesssim_{\mathsf{t}} (\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathsf{wr}'''_1(\mathbf{return}\ E); \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
(\mathbf{return}\ E; \mathbf{noret}, \sigma_a \uplus \{\mathtt{l} \leadsto \_\}, \kappa') \lesssim_{\mathsf{t}} (\mathbf{return}\ E; \mathbf{noret}, \sigma_a, \kappa) \\
\text{where } \kappa' = (s_l \uplus \{\mathtt{u1} \leadsto \_, \mathtt{u2} \leadsto \_\}, x, C')\ ,\ \text{if } \kappa = (s_l, x, C')
\end{array}
$$

By case analysis and operational semantics, we can prove the following property of the simulation.

If $(W_B, \mathcal{S}_B) \lesssim (W_A, \mathcal{S}_A)$, then the following hold.
(1) If $(W_B, \mathcal{S}_B) \xmapsto{e_B} \mathbf{abort}$, then
there exists T_A such that $(W_A, \mathcal{S}_A) \xmapsto{T_A}{}^* \mathbf{abort}$ and $e_B = \mathsf{get\_obsv}(T_A)$.
(2) If $(W_B, \mathcal{S}_B) \xmapsto{e_B} (W'_B, \mathcal{S}'_B)$, then
there exist T_A, W'_A and $\mathcal{S}'_A$ such that $(W_A, \mathcal{S}_A) \xmapsto{T_A}{}^* (W'_A, \mathcal{S}'_A)$, $\mathsf{get\_obsv}(e_B) = \mathsf{get\_obsv}(T_A)$ and $(W'_B, \mathcal{S}'_B) \lesssim (W'_A, \mathcal{S}'_A)$.
(B.24)

With (B.24), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{O}[\![W_B, \mathcal{S}_B]\!]$.

If $(W_B, \mathcal{S}_B) \lesssim (W_A, \mathcal{S}_A)$ and $T \in \mathcal{O}[\![W_B, \mathcal{S}_B]\!]$, then $T \in \mathcal{O}[\![W_A, \mathcal{S}_A]\!]$.
(B.25)

Since

$$(\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a \uplus \{\mathtt{l} \leadsto 0\}, \circledcirc)) \lesssim (\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc))\ ,$$

from (B.25), we are done. □

Proof of (B.23). We want to show: for any n, $C_1, \ldots, C_n$, σ_c and σ_a,

$$
\begin{array}{l}
\forall T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge (\sigma_a(\mathtt{l}) = 0) \\
\implies \mathsf{deadlock\text{-}free}(T)\ .
\end{array}
$$

It is reduced to the following.

$$
\begin{array}{l}
\forall T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge (\sigma_a(\mathtt{l}) = 0) \wedge \mathsf{fair}(T) \wedge \neg\mathsf{abt}(T) \\
\implies \mathsf{prog\text{-}s}(T)\ .
\end{array}
$$

- If $|T| \neq \omega$, we know $\mathsf{prog\text{-}s}(T)$ must hold.
- If $|T| = \omega$, suppose $\mathsf{prog\text{-}s}(T)$ does not hold. Then there exist i and e such that $e \in \mathsf{pend\_inv}(T(1..i))$ and $\forall j.\ j > i \implies \neg\mathsf{is\_ret}(T(j))$.
Suppose the execution generating such T is:

$$(W_0, \mathcal{S}_0) \xmapsto{T(1)} (W_1, \mathcal{S}_1) \xmapsto{T(2)} (W_2, \mathcal{S}_2) \xmapsto{T(3)} \ldots$$

Here we write W_0 for $\mathbf{let}\ \mathsf{wr}_1(\Gamma)\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n$ and $\mathcal{S}_0$ for $(\sigma_c, \sigma_a, \circledcirc)$.
Suppose $\mathsf{tid}(e) = \mathsf{t}$ and $e = T(i_0)$. We know $\neg\exists j.\ (j > i_0 \wedge \mathsf{match}(e, T(j)))$.
Since $\mathsf{fair}(T)$, we know $|(T|_{\mathsf{t}})| = \omega$. From the operational semantics, we know there exist $i_1, i_2, \ldots$ (an infinite number) such that $i_0 < i_1 < i_2 < \ldots$, $T(i_1) = T(i_2) = \ldots = (\mathsf{t}, \mathbf{obj})$, and the code of thread t in $W_{i_1-1}, W_{i_1}, W_{i_2-1}, W_{i_2}, \ldots$ is all the same, which is either $\mathsf{wr}'_1(\langle C\rangle); \mathsf{wr}_1(\mathbf{return}\ E); \mathbf{noret}$ or $\mathsf{wr}'_1(\mathbf{return}\ E); \mathbf{noret}$.
Suppose the lock $\mathtt{l}$ has the values $\mathsf{t}_1, \mathsf{t}_2, \ldots$ in the states $\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \ldots$. We know $\mathsf{t}_1 \neq 0$, $\mathsf{t}_2 \neq 0, \ldots$. Since there are only n threads, we know there exist $j, j_1, j_2, \ldots$ (an infinite number) such that $\mathsf{t}_j = \mathsf{t}_{j_1} = \mathsf{t}_{j_2} = \ldots$. In other words, thread t_j holds the lock infinitely often in the execution.
Since $\mathsf{fair}(T)$, we know $|(T|_{\mathsf{t}_j})| = \omega$. By the operational semantics, we know t_j must return infinitely often.
So we get a contradiction. Thus $\mathsf{prog\text{-}s}(T)$ holds.

Thus we are done. □

Proofs of (B.19). The key is to show the following (B.26).

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_1$ and σ_a such that $(\sigma_o, \sigma_1) \models P$ and $\sigma_a = \sigma_1 \uplus \{\mathtt{l} \rightsquigarrow 0\}$,
if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, then there exists T_b such that
$(\lfloor \textbf{let}\ \mathsf{wr_1}(\Gamma)\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_b}{\longmapsto}{}^{\omega} \cdot$, $\mathsf{fair}(T_b)$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_b)$.

(B.26)

Following the proof of (B.20), we get:

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and σ_1 such that $(\sigma_o, \sigma_1) \models P$,
if $(\lfloor \textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$, then there exists T_a such that
$(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_1, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$ and $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$.

(B.27)

Since $\mathsf{fair}(T)$, we can prove that T_a satisfies the following $\mathsf{cltfair}$ in Definition 67. Since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, from $\mathsf{prog\text{-}s}(T)$, we know $\mathsf{prog\text{-}s}(T_a)$ holds. By the following Lemma 68, we can construct the execution of **let** $\mathsf{wr_1}(\Gamma)$ **in** $C_1 \parallel \ldots \parallel C_n$ generating the trace T_b such that $\mathsf{fair}(T_b)$ and $T_a \backslash (\_, \textbf{obj}) = T_b \backslash (\_, \textbf{obj})$ hold. Thus we are done.

Definition 67. $\mathsf{cltfair}(T_a)$ iff

$$|T_a| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T_a)].\ |(T_a|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T_a|_{\mathsf{t}}) = (\mathsf{t}, \textbf{term}) \\ \vee\ \mathsf{last}(T_a|_{\mathsf{t}}) = (\mathsf{t}, \textbf{obj}) \vee \exists f, n.\ \mathsf{last}(T_a|_{\mathsf{t}}) = (\mathsf{t}, f, n)$$

Lemma 68. If $(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$, $\mathsf{cltfair}(T_a)$ and $\mathsf{prog\text{-}s}(T_a)$, then there exists T_b such that $(\lfloor \textbf{let}\ \mathsf{wr_1}(\Gamma)\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{l \rightsquigarrow 0\}, \circledcirc)) \overset{T_b}{\longmapsto}{}^{\omega} \cdot$, $\mathsf{fair}(T_b)$ and $T_a \backslash (\_, \textbf{obj}) = T_b \backslash (\_, \textbf{obj})$.

Proof. We construct a simulation relation $\lesssim$ that satisfies the following property.

If $T \models (W_a, \mathcal{S}_a) \lesssim (W_b, \mathcal{S}_b)$ and $(W_a, \mathcal{S}_a) \overset{e_a}{\longmapsto} (W'_a, \mathcal{S}'_a)$, then
there exist T_b, W'_b and $\mathcal{S}'_b$ such that $(W_b, \mathcal{S}_b) \overset{T_b}{\longmapsto}{}^{+} (W'_b, \mathcal{S}'_b)$,
$T :: e_a \models (W'_a, \mathcal{S}'_a) \lesssim (W'_b, \mathcal{S}'_b)$, $e_a \backslash (\_, \textbf{obj}) = T_b \backslash (\_, \textbf{obj})$ and
if $\mathsf{is\_ret}(e_a)$ holds, then $\forall e.\ e \in \mathsf{pend\_inv}(T) \Rightarrow \exists i.\ \mathsf{tid}(e) = \mathsf{tid}(T_b(i))$.

(B.28)

The simulation is established as follows.

$$\begin{array}{l}
T \models (\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_a, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})) \\
\quad \lesssim (\textbf{let}\ \mathsf{wr_1}(\Gamma)\ \textbf{in}\ C'_1 \parallel \ldots \parallel C'_n, (\sigma_c, \sigma'_a, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\})) \\
\qquad\qquad\qquad\qquad \text{if } \forall i.\ (C_i, \sigma_a, \kappa_i) \lesssim_i (C'_i, \sigma'_a, \kappa'_i) \\[1em]
(C, \sigma_a, \circ) \lesssim_{\mathsf{t}} (C, \sigma_a \uplus \{l \rightsquigarrow 0\}, \circ) \\
(\langle C \rangle; \textbf{return}\ E; \textbf{noret}, \sigma_a, \kappa) \lesssim_{\mathsf{t}} (\mathsf{wr_1}(\langle C \rangle); \mathsf{wr_1}(\textbf{return}\ E); \textbf{noret}, \sigma_a \uplus \{\mathtt{l} \rightsquigarrow 0\}, \kappa') \\
(\textbf{return}\ E; \textbf{noret}, \sigma_a, \kappa) \lesssim_{\mathsf{t}} (\mathsf{wr_1}(\textbf{return}\ E); \textbf{noret}, \sigma_a \uplus \{\mathtt{l} \rightsquigarrow 0\}, \kappa') \\
\text{where } \kappa' = (s_l \uplus \{\mathtt{u1} \rightsquigarrow \_, \mathtt{u2} \rightsquigarrow \_\}, x, C')\ ,\ \text{if } \kappa = (s_l, x, C')
\end{array}$$

We can prove (B.28) by case analysis and operational semantics. The idea is to execute the lock instructions of the threads which are pending just after the lock instruction of the thread that returns. Then if $\mathsf{is\_ret}(e_a)$ holds, the pending threads in T must have been executed in the execution of $(W_b, \mathcal{S}_b)$.

With (B.28), we can prove the following by co-induction over the event trace T_a generated in $(W_a, \mathcal{S}_a) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$.

If $\epsilon \models (W_a, \mathcal{S}_a) \lesssim (W_b, \mathcal{S}_b)$, $(W_a, \mathcal{S}_a) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{prog\text{-}s}(T_a)$, then
there exists T_b such that $(W_b, \mathcal{S}_b) \overset{T_b}{\longmapsto}{}^{\omega} \cdot$, $T_a \backslash (\_, \textbf{obj}) = T_b \backslash (\_, \textbf{obj})$ and
$\forall e.\ e \in \mathsf{pend\_inv}(T_b) \Rightarrow \forall i.\ \exists j > i.\ \mathsf{tid}(T_b(j)) = \mathsf{tid}(e)$.

(B.29)

Since

$$\epsilon \models (\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \lesssim (\lfloor \textbf{let}\ \mathsf{wr_1}(\Gamma)\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{\mathtt{l} \rightsquigarrow 0\}, \circledcirc))\ ,$$

from (B.29), we know there exists T_b such that

$$(\lfloor \textbf{let}\ \mathsf{wr_1}(\Gamma)\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a \uplus \{\mathtt{l} \rightsquigarrow 0\}, \circledcirc)) \overset{T_b}{\longmapsto}{}^{\omega} \cdot,$$

$T_a \backslash (\_, \textbf{obj}) = T_b \backslash (\_, \textbf{obj})$ and $\forall e.\ e \in \mathsf{pend\_inv}(T_b) \Rightarrow \forall i.\ \exists j > i.\ \mathsf{tid}(T_b(j)) = \mathsf{tid}(e)$.
Since $(\lfloor \textbf{let}\ \Gamma\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{cltfair}(T_a)$, we know $\mathsf{fair}(T_b)$ holds. Thus we are done. ☐

C. Starvation-free examples

C.1 Counter with ticket lock

Our logic LiLi can be applied to verify simple objects using a single lock to protect all the object data. As examples, we verify the counter object using a ticket lock (see below), or a queue lock (see Secs. C.2, C.3 and C.4), or a test-and-set lock (see Sec. D.1).

Fig. 27 shows the concrete implementation of the counter `sfInc` with a ticket lock, where we write the auxiliary code in red. We implicitly assume every instruction that accesses the shared state is in an atomic block.

As we explained in Sec. 2, `sfInc` in Fig. 1(c) is linearizable with respect to the abstract atomic operation `INC` because `sfInc` ensures mutually exclusive access to `x`. When a thread acquires the lock, the shared resource `x` at both the concrete and the abstract sides is transferred to the thread's local state. `sfInc` ensures starvation-freedom because the threads which are currently requesting the lock constitute a queue.

We introduce some auxiliary structures to facilitate the proof. As shown in Fig. 27, we introduce the explicit tickets $\mathtt{ticket}_0$, $\mathtt{ticket}_1$, ... Each $\mathtt{ticket}_i$ records the ID of the unique thread which gets the ticket number i. We use `cid` to record the current thread ID. The explicit connection from the ticket numbers to the thread IDs gives us the knowledge about the queue of the threads requesting the lock.

Besides, each ticket i is accompanied with a waiting bit $\mathtt{wait}_i$ to indicate whether the thread $\mathtt{ticket}_i$ is requesting the lock or has entered its critical section. It helps describe the ownership transfer over the resource `x`. Initially $\mathtt{wait}_i$ is false. It is set to true when the thread gets its ticket at line 2. After the loop, $\mathtt{wait}_i$ is reset (line 7) to transfer the shared resource `x` to the thread so that the thread can freely access `x` in the critical section (line 8).

Fig. 28 defines the precise invariant I which determines the boundaries of shared states. A shared state contains the lock (with $\mathtt{owner} = n_1$ and $\mathtt{next} = n_2$), the wait bits (waits) and the protected $\mathsf{resource}$ if $\mathtt{wait}_{\mathtt{owner}}$ holds. Here $\mathsf{resource}$ requires `x` to be the same at the concrete and the abstract sides. $\mathsf{lock}(\mathit{tl}, n_1, n_2)$ contains the auxiliary `tickets` in addition to `owner` and `next`, where *tl* (hidden in the definition of I) is the list of the threads $\mathtt{ticket}_{\mathtt{owner}}, \ldots, \mathtt{ticket}_{\mathtt{next}-1}$. We also use $\mathsf{locked}(\mathit{tl}, n_1, n_2)$ for the case when *tl* is not empty. That is, the lock is acquired by the first thread in *tl*, while the other threads in *tl* are waiting for the lock in order. Besides, we use $\mathsf{lockIrr}_{\mathtt{t}}(\mathit{tl}, n_1, n_2)$ short for $\mathsf{lock}(\mathit{tl}, n_1, n_2) \wedge (\mathtt{t} \notin \mathit{tl})$. That is, the thread `t` is irrelevant to the lock: it does not acquire or request the lock. The precondition P is stronger than I. For thread `t`, $P_{\mathtt{t}}$ requires the lock to satisfy $\exists \mathit{tl}.\ \mathsf{lockIrr}_{\mathtt{t}}(\mathit{tl}, n_1, n_2)$.

The guarantee condition G in Fig. 28 defines the atomic actions of a thread `t`. $\mathit{ReqLock}_{\mathtt{t}}$ adds the thread `t` at the end of *tl* of the threads requesting the lock and increments `next`. It corresponds to line 2 of Fig. 27. $\mathit{AcqResource}_{\mathtt{t}}$ transfers the $\mathsf{resource}$ to the thread `t` when it has acquired the lock, corresponding to line 7 of Fig. 27. $\mathit{RelLock}_{\mathtt{t}}$ releases the lock and transfers the $\mathsf{resource}$ back to the shared state. The thread `t` which currently holds the lock is dequeued from the list *tl* and `owner` is incremented. It corresponds to line 9 of Fig. 27. The rely condition R include all the atomic actions made by the environment threads.

Next we define the definite action $\mathcal{D}$ in Fig. 28. As we explained in Sec. 2, the thread `t`'s definite action $\mathcal{D}_{\mathtt{t}}$ requires that whenever `t` holds a lock with $\mathtt{owner} = n_1$ (specified by $dp_{\mathtt{t}}(n_1)$), it should eventually release the lock by incrementing `owner` to $n_1 + 1$ (specified by $dq_{\mathtt{t}}(n_1)$). Note that $dp_{\mathtt{t}}(n_1)$ allows the $\mathsf{resource}$ to be either in the shared state or in the thread `t`'s local state. The former case means that the thread `t` has not executed line 7 but its loop at lines 4-6 must terminate. Besides, $dp_{\mathtt{t}}(n_1)$ is stable under the environment which may enqueue more threads into the list *tl* of waiting threads. We can prove the side conditions about well-formedness of specifications in the `OBJ` rule in Fig. 9 holds.

Fig. 29 shows the proof outline. To verify the loop at lines 4-6, we first define J and Q in Fig. 28. Both J and Q are stronger than I. For thread `t`, $J_{\mathtt{t}}$ says `t` is requesting the lock. Here $\mathsf{tlocked}_{\mathit{tl}_1,\mathtt{t},\mathit{tl}_2}(n_1, X, n_2)$ says (1) `t` takes the ticket number X, which satisfies $n_1 = \mathtt{owner} \leq X < \mathtt{next} = n_2$, and (2) the threads requesting the lock before and after `t` constitute the lists tl_1 and tl_2 respectively. Then, the first thread of the whole concatenated list $\mathit{tl}_1 :: \mathtt{t} :: \mathit{tl}_2$ holds the lock. $Q_{\mathtt{t}}$ specifies the case when tl_1 is empty (thus $X = \mathtt{owner}$). The loop terminates when Q holds. We also strengthen the guarantee of the loop to $G' \overset{\text{def}}{=} [I]$, the identity transitions. We have $\mathsf{Sta}(\{J, Q\}, G' \vee R)$.

Next we define f and prove $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$. The metric function f maps each shared state $\mathfrak{S}$ to the value of $(X - \mathtt{owner})$ at that state. Here X is a logical variable recording the ticket number of the current thread (i.e., $X = \mathtt{i}$ holds). Since J ensures $\mathtt{owner} \leq X$, we can use the usual order on natural numbers as the associated well-founded order. The condition $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ holds due to the following reasons:

(1) Each action in R or G' either increases `owner` or keeps `owner` unchanged.

(2) Either Q holds, or $\mathtt{owner} < X$ and some environment thread $\mathtt{t}'$ acquires the lock. In the latter case, the value of $(X - \mathtt{owner})$ decreases when $\mathtt{t}'$ releases the lock. That is, the metric decreases after a definite action made by the environment.

Finally we conclude the full correctness of `sfInc` with respect to the atomic `INC` under fair scheduling. By Theorem 2 (Soundness), we get linearizability and starvation-freedom of `sfInc`.

```
initialize(){  owner := 0; next := 0;  }

sfInc(){
 local i, o, r;
 <i := getAndInc(next); ticket_i := cid; wait_i := true>;
 o := owner;
 while (i != o) {
   o := owner;
 }
 <wait_i := false>;
 r := x; x := r + 1;
 owner := i + 1;
}
```

Figure 27. Counter `sfInc` with a ticket lock (auxiliary code in red).

$tl \ ::= \ \epsilon \ | \ \mathtt{t} :: tl$

$\mathsf{list2set}(\epsilon) \stackrel{\text{def}}{=} \emptyset$

$\mathsf{list2set}(\mathtt{t}::tl) \stackrel{\text{def}}{=} \{\mathtt{t}\} \cup \mathsf{list2set}(tl)$

$I \stackrel{\text{def}}{=} \exists n_1, n_2.\, \mathsf{lock}(n_1, n_2) * (\mathtt{wait}_{n_1} * \mathsf{resource} \vee \neg\mathtt{wait}_{n_1}) * \mathsf{waits}(n_1, n_2)$

$\mathsf{resource} \stackrel{\text{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad \mathsf{lres} \stackrel{\text{def}}{=} (\mathtt{x} = \_)$

$\mathsf{waits}(n_1, n_2) \stackrel{\text{def}}{=} (\circledast_{0 \le i < n_1 \vee i \ge n_2} (\neg\mathtt{wait}_i)) * (\circledast_{n_1 < i < n_2} \mathtt{wait}_i)$

$\mathsf{lock}(n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\, \mathsf{lock}(tl, n_1, n_2)$

$\mathsf{lock}(tl, n_1, n_2) \stackrel{\text{def}}{=} ((\mathtt{owner} = n_1) * (\mathtt{next} = n_2) \wedge (n_1 \le n_2)) * \mathsf{tickets}(0, n_1) * \mathsf{tickets}(tl, n_1, n_2) * \mathsf{tickets\_new}(n_2)$

$\mathsf{tickets}(n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\, \mathsf{tickets}(tl, n_1, n_2)$

$\mathsf{tickets}(tl, n_1, n_2) \stackrel{\text{def}}{=}$
$\quad (tl = \epsilon) \wedge (n_1 = n_2) \wedge \mathsf{emp} \ \vee \ \exists \mathtt{t}, tl'.\, (tl = \mathtt{t}::tl') \wedge (\mathtt{t} \notin \mathsf{list2set}(tl')) \wedge (\mathtt{ticket}_{n_1} = \mathtt{t}) * \mathsf{tickets}(tl', n_1 + 1, n_2)$

$\mathsf{tickets\_new}(n_2) \stackrel{\text{def}}{=} (\circledast_{i \ge n_2} \mathtt{ticket}_i = -1)$

$P_{\mathtt{t}} \stackrel{\text{def}}{=} \exists n_1, n_2.\, \mathsf{lock\_irr}_{\mathtt{t}}(n_1, n_2) * (\mathtt{wait}_{n_1} * \mathsf{resource} \vee \neg\mathtt{wait}_{n_1}) * \mathsf{waits}(n_1, n_2)$

$\mathsf{lock\_irr}_{\mathtt{t}}(n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\, \mathsf{lock\_irr}_{\mathtt{t}}(tl, n_1, n_2) \qquad \mathsf{lock\_irr}_{\mathtt{t}}(tl, n_1, n_2) \stackrel{\text{def}}{=} \mathsf{lock}(tl, n_1, n_2) \wedge (\mathtt{t} \notin \mathsf{list2set}(tl))$

$R_{\mathtt{t}} \stackrel{\text{def}}{=} \bigvee_{\mathtt{t}' \neq \mathtt{t}} G_{\mathtt{t}'}$

$G_{\mathtt{t}} \stackrel{\text{def}}{=} (\mathit{ReqLock}_{\mathtt{t}} \vee \mathit{AcqResource}_{\mathtt{t}} \vee \mathit{RelLock}_{\mathtt{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{ReqLock}_{\mathtt{t}} \stackrel{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\, (\mathsf{lock\_irr}_{\mathtt{t}}(tl, n_1, n_2) * (\neg\mathtt{wait}_{n_2})) \ltimes (\mathsf{lock}(tl::\mathtt{t}, n_1, n_2 + 1) * \mathtt{wait}_{n_2})$

$\mathit{AcqResource}_{\mathtt{t}} \stackrel{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\, (\mathsf{lock}(\mathtt{t}::tl, n_1, n_2) * \mathtt{wait}_{n_1} * \mathsf{resource}) \ltimes (\mathsf{lock}(\mathtt{t}::tl, n_1, n_2) * (\neg\mathtt{wait}_{n_1}))$

$\mathit{RelLock}_{\mathtt{t}} \stackrel{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\, (\mathsf{lock}(\mathtt{t}::tl, n_1, n_2)) \ltimes (\mathsf{lock\_irr}_{\mathtt{t}}(tl, n_1 + 1, n_2) * \mathsf{resource})$

$\mathcal{D}_{\mathtt{t}} \stackrel{\text{def}}{=} \forall n_1.\, dp_{\mathtt{t}}(n_1) \leadsto dq_{\mathtt{t}}(n_1)$

$dp_{\mathtt{t}}(n_1) \stackrel{\text{def}}{=} \exists tl, n_2.\, \mathsf{lock}(\mathtt{t}::tl, n_1, n_2) * \mathsf{true} \wedge I$

$dq_{\mathtt{t}}(n_1) \stackrel{\text{def}}{=} \exists n_2.\, \mathsf{lock\_irr}_{\mathtt{t}}(n_1 + 1, n_2) * \mathsf{true} \wedge I$

Figure 28. Invariant, precondition, rely/guarantee and definite action of counter with ticket lock.

$$\begin{aligned}&\mathsf{tlocked}_{tl_1,\mathtt{t},tl_2}(n_1,n,n_2) \overset{\text{def}}{=}\\ &\quad (\mathsf{list2set}(tl_1)\cap \mathsf{list2set}(\mathtt{t}::tl_2)=\emptyset)\wedge((\mathtt{owner}=n_1)*(\mathtt{next}=n_2)\wedge(n_1\le n<n_2))\\ &\quad *\,\mathsf{tickets}(0,n_1)*\mathsf{tickets}(tl_1,n_1,n)*\mathsf{tickets}(\mathtt{t}::tl_2,n,n_2)*\mathsf{tickets\_new}(n_2)\end{aligned}$$

$P_0(n_1,n,n_2) \overset{\text{def}}{=} \exists tl_1.\ P_0(tl_1,n_1,n,n_2)$ $\qquad$ $P_3(n_1,n,n_2) \overset{\text{def}}{=} \exists \mathtt{t}', tl_1.\ P_0(\mathtt{t}'::tl_1,n_1,n,n_2)$

$P_0(tl_1,n_1,n,n_2) \overset{\text{def}}{=} \exists tl_2.\ \mathsf{tlocked}_{tl_1,\mathtt{t},tl_2}(n_1,n,n_2)*(\mathtt{wait}_{n_1}*\mathsf{resource}\vee(\neg\mathtt{wait}_{n_1})\wedge(n_1<n))*\mathsf{waits}(n_1,n_2)$

$P_1(n_1,n_2) \overset{\text{def}}{=} \exists tl.\ \mathsf{lock}(\mathtt{t}::tl,n_1,n_2)*\mathtt{wait}_{n_1}*\mathsf{resource}*\mathsf{waits}(n_1,n_2)$

$P_2(n_1,n_2) \overset{\text{def}}{=} \exists tl.\ \mathsf{lock}(\mathtt{t}::tl,n_1,n_2)*(\neg\mathtt{wait}_{n_1})*\mathsf{waits}(n_1,n_2)$

```
inc():
 1  local i, o, r;
   { $P \wedge \mathsf{arem}(\mathtt{INC})$ }
 2  < i := getAndInc(next); ticket_i := cid; wait_i := true; >
   { $\exists n_1,n,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge\mathsf{arem}(\mathtt{INC})$ }
   { $\exists n_1,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge\mathsf{arem}(\mathtt{INC})$ }
 3  o := owner;
   { $\exists n_1,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge(\mathtt{o}\le n_1)\wedge\mathsf{arem}(\mathtt{INC})$ }
   { $\exists n_1,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge(\mathtt{o}\le n_1)\wedge\mathsf{arem}(\mathtt{INC})\wedge\Diamond(n-\mathtt{o})$ }
 4  while (i != o) {
 5    o := owner;
 6  }
   { $\exists n_1,n_2.\ P_1(n_1,n_2)\wedge(\mathtt{i}=n_1)\wedge\mathsf{arem}(\mathtt{INC})$ }
 7   <wait_i := false>;
   { $\mathsf{resource}*(\exists n_1,n_2.\ P_2(n_1,n_2))\wedge(\mathtt{i}=n_1)\wedge\mathsf{arem}(\mathtt{INC})$ }
 8  r := x; x := r + 1;
   { $(\mathtt{x}=\mathtt{X}+1)*(\exists n_1,n_2.\ P_2(n_1,n_2))\wedge(\mathtt{i}=n_1)\wedge\mathsf{arem}(\mathtt{INC})$ }
 9  owner := i + 1;
   { $P \wedge \mathsf{arem}(\mathbf{skip})$ }
```

Here the loop at lines 4-6 is verified as follows. Let

$p' \overset{\text{def}}{=} \exists n_1,n_2.\ (P_1(n,n_2)\wedge\Diamond(n-(\mathtt{o}+1))\vee P_3(n_1,n,n_2)\wedge\Diamond(n-\mathtt{o}))\wedge(\mathtt{o}\le n_1\le n=\mathtt{i})\wedge(\mathtt{o}\ne\mathtt{i})\wedge\mathsf{arem}(\mathtt{INC})$

$J \overset{\text{def}}{=} \exists n_1,n_2.\ P_0(n_1,n,n_2)$

$Q \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D})$

$G' \overset{\text{def}}{=} [I]$

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

$$\begin{aligned}&(\exists n_1,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge(\mathtt{o}\le n_1)\wedge(\mathtt{i}\ne\mathtt{o})\wedge\mathsf{arem}(\mathtt{INC})\wedge\Diamond(n-\mathtt{o})\wedge Q*\mathsf{true})\\ &\Longrightarrow(\exists n_2.\ P_1(n,n_2)\wedge(\mathtt{i}=n)\wedge(\mathtt{o}<n)\wedge\mathsf{arem}(\mathtt{INC})\wedge\Diamond(n-(\mathtt{o}+1)))*(\Diamond(1)\wedge\mathsf{emp})\end{aligned}$$

```
     { $p'$ }
     { $\exists n_1,n_2.\ (P_1(n,n_2)\wedge\Diamond(n-(\mathtt{o}+1))\vee P_3(n_1,n,n_2)\wedge\Diamond(n-\mathtt{o}))\wedge(\mathtt{o}\le n_1\le n=\mathtt{i})\wedge(\mathtt{o}\ne\mathtt{i})\wedge\mathsf{arem}(\mathtt{INC})$ }
5     <o := owner>;
     { $\exists n_1,n_2.\ P_0(n_1,n,n_2)\wedge(\mathtt{i}=n)\wedge(\mathtt{o}\le n_1)\wedge\mathsf{arem}(\mathtt{INC})\wedge\Diamond(n-\mathtt{o})$ }
     { $p$ }
```

We can prove:

(1) $J\Rightarrow I;\ Q\Rightarrow I;\ \mathsf{Sta}(J,G'\vee R)$.

(2) $J\Rightarrow(R,G':\mathcal{D}\xrightarrow{f}Q)$ (where picking $\mathcal{D}'$ in the WHL rule as $\mathcal{D}$).

Figure 29. Proof outline. We prove: $\mathcal{D},R,G,I\vdash\{P\wedge\mathsf{arem}(\mathtt{INC})\}C\{P\wedge\mathsf{arem}(\mathbf{skip})\}$.

```
initialize(){ tail := 0; flag[0] := true; flag[1..TNUM-1] := false; }

inc(){
 1  local i, b, r;
 2  <i := getAndInc(tail) % TNUM; queue[i] := cid; wait[i] := true>;
 3  b := flag[i];
 4  while (!b) {
 5    b := flag[i];
 6  }
 7  <wait[i] := false>;
 8  r := x; x := r + 1;
 9  flag[i] := false;
10  <flag[(i + 1) % TNUM] := true; queue[i] := -1>;
}
```

Figure 30. Counter with Anderson array-based lock.

C.2 Counter with Anderson array-based queue lock

In this section, we verify the counter implementation with the Anderson array-based queue lock is linearizable with respect to the atomic operation `INC` and is starvation-free. Fig. 30 shows the concrete implementation with auxiliary code (in red). Suppose the maximum total number of threads is `TNUM`. Anderson array-based lock algorithm [13] uses a boolean `flag` array which contains `TNUM` slots and a `tail` pointer which points to the next available slot in the array. To acquire the lock, each thread reads the current value of `tail` and atomically increments `tail` (line `2`). Then the thread spins until the flag at its slot becomes `true` (lines `3-6`). If `flag[i]` is `true`, then the thread with slot `i` has the permission to acquire the lock. To release the lock, the thread sets the flag at its slot to `false` (line `9`) and then sets the flag at the next slot to `true` (line `10`) to let the next thread acquire the lock.

To verify the code, we introduce an auxiliary array `queue` which contains `TNUM` slots (just like the `flag` array in the original code). The `queue` array records the thread ID `t` if the corresponding slot at the `flag` array is held by the thread `t`. For an available slot i (i.e., which is not held by any thread), $\texttt{queue}[i]$ is -1. Thus we can know from the `queue` array the list of threads requesting the lock. Similarly to the previous example of the counter with ticket lock, we also give each slot i a waiting bit $\texttt{wait}[i]$ to indicate whether the thread $\texttt{queue}[i]$ is requesting the lock or has entered the critical section.

Fig. 31 shows the full definitions of the precise invariant, the precondition, the rely and guarantee conditions and the definite action. The definitions are similar the the corresponding ones for the counter with ticket lock. The most interesting predicate in the definition of the invariant I is $\texttt{queue}(tl, n_1, n_2, sz)$, which gives the queue tl of the threads requesting the lock, and the start and the end slots n_1 and n_2 in the arrays. The definition of $\texttt{queue}(tl, n_1, n_2, sz)$ contains two cases: the first case says the queue of the threads does not step over the boundary of the arrays (thus $n_1 \leq n_2$), while the second case is the opposite. When $n_1 = n_2$, the first case means the queue is empty (no threads are requesting the lock), while the second case means the queue is full (all the `TNUM` threads are requesting the lock).

Note that as shown in the definition of the guaranteed actions *SetFlag* and *RelLock*, we view line `10` as the action of releasing the lock and giving up the slot of the current thread. Line `9` is to reset the slot of the current thread, which is viewed as an action happened when the thread is holding the lock.

Fig. 32 shows the proof outline. The proofs are similar to the proofs of the counter with ticket lock.

$$tl ::= \epsilon \mid \mathsf{t} :: tl$$

$$\mathsf{list2set}(\epsilon) \overset{\text{def}}{=} \emptyset$$

$$\mathsf{list2set}(\mathsf{t} :: tl) \overset{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(tl)$$

$$I \overset{\text{def}}{=} \exists sz, n_1, n_2.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(n_1, n_2, sz) * (\mathtt{wait}[n_1] * \mathsf{resource} \vee \neg\mathtt{wait}[n_1]) * \mathsf{waits}(n_1, n_2, sz)$$

$$\mathsf{resource} \overset{\text{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad \mathsf{lres} \overset{\text{def}}{=} (\mathtt{x} = \_)$$

$$\mathsf{lock}(n_1, n_2, sz) \overset{\text{def}}{=} \exists tl, b.\ \mathsf{lock}(tl, n_1, n_2, sz, b) \qquad \mathsf{lock\_irr}_{\mathsf{t}}(n_1, n_2, sz) \overset{\text{def}}{=} \exists tl, b.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, n_1, n_2, sz, b)$$

$$\mathsf{lock}(tl, n_1, n_2, sz, b) \overset{\text{def}}{=} \mathsf{queue}(tl, n_1, n_2, sz) * (\mathtt{tail}\%sz = n_2) * \mathsf{fflags}(n_1, sz) * (\mathtt{flag}[n_1] = b) \wedge (b \vee (\neg b) \wedge (n_1 < n_2))$$

$$\mathsf{lock\_irr}_{\mathsf{t}}(tl, n_1, n_2, sz, b) \overset{\text{def}}{=} \mathsf{lock}(tl, n_1, n_2, sz, b) \wedge (\mathsf{t} \notin \mathsf{list2set}(tl))$$

$$\mathsf{fflags}(n_1, sz) \overset{\text{def}}{=} (\circledast_{0 \le i < n_1 \wedge n_1 < i < sz}(\neg\mathtt{flag}[i]))$$

$$\begin{aligned}
&\mathsf{queue}(tl, n_1, n_2, sz) \overset{\text{def}}{=} \\
&\quad (0 \le n_1 \le n_2 < sz) \wedge \mathsf{free}(0, n_1) * \mathsf{thrds}(tl, n_1, n_2) * \mathsf{free}(n_2, sz) \\
&\quad \vee (0 \le n_2 \le n_1 < sz) \wedge \exists tl_1, tl_2.\ (tl = tl_1 :: tl_2) \wedge \mathsf{thrds}(tl_2, 0, n_2) * \mathsf{free}(n_2, n_1) * \mathsf{thrds}(tl_1, n_1, sz)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{thrds}(tl, n, n') \overset{\text{def}}{=} \\
&\quad (tl = \epsilon) \wedge (n = n') \wedge \mathsf{emp} \ \vee\ \exists \mathsf{t}, tl'.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (\mathtt{queue}[n] = \mathsf{t}) * \mathsf{thrds}(tl', n+1, n')
\end{aligned}$$

$$\mathsf{free}(n, n') \overset{\text{def}}{=} (\circledast_{n \le i < n'} \mathtt{queue}[i] = -1)$$

$$\begin{aligned}
&\mathsf{waits}(n_1, n_2, sz) \overset{\text{def}}{=} \\
&\quad (0 \le n_1 \le n_2 < sz) \wedge (\circledast_{0 \le i < n_1 \vee n_2 \le i < sz}(\neg\mathtt{wait}[i])) * (\circledast_{n_1 < i < n_2} \mathtt{wait}[i]) \\
&\quad \vee (0 \le n_2 < n_1 < sz) \wedge (\circledast_{0 \le i < n_2} \mathtt{wait}[i]) * (\circledast_{n_2 \le i < n_1}(\neg\mathtt{wait}[i])) * (\circledast_{n_1 < i < sz} \mathtt{wait}[i])
\end{aligned}$$

$$P_{\mathsf{t}} \overset{\text{def}}{=} \exists sz, n_1, n_2.\ (\mathtt{TNUM} = sz) * \mathsf{lock\_irr}_{\mathsf{t}}(n_1, n_2, sz) * (\mathtt{wait}[n_1] * \mathsf{resource} \vee \neg\mathtt{wait}[n_1]) * \mathsf{waits}(n_1, n_2, sz) \wedge (1 \le \mathsf{t} \le sz)$$

$$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \ne \mathsf{t}} G_{\mathsf{t}'}$$

$$\begin{aligned}
&G_{\mathsf{t}} \overset{\text{def}}{=} \\
&\quad \exists sz.\ (\mathit{ReqLock}_{\mathsf{t}}(sz) \vee \mathit{AcqResource}_{\mathsf{t}}(sz) \vee \mathit{SetFlag}_{\mathsf{t}}(sz) \vee \mathit{RelLock}_{\mathsf{t}}(sz) \vee \mathsf{Id}) \\
&\quad * [(\mathtt{TNUM} = sz) \wedge (1 \le \mathsf{t} \le sz)] * \mathsf{Id} \wedge (I \ltimes I)
\end{aligned}$$

$$\begin{aligned}
&\mathit{ReqLock}_{\mathsf{t}}(sz) \overset{\text{def}}{=} \\
&\quad \exists tl, n_1, n_2, b.\ ((\mathsf{lock\_irr}_{\mathsf{t}}(tl, n_1, n_2, sz, b) * (\neg\mathtt{wait}[n_2])) \ltimes (\mathsf{lock}(tl :: \mathsf{t}, n_1, (n_2+1)\%sz, sz, b) * \mathtt{wait}[n_2]))
\end{aligned}$$

$$\begin{aligned}
&\mathit{AcqResource}_{\mathsf{t}}(sz) \overset{\text{def}}{=} \\
&\quad \exists tl, n_1, n_2.\ (\mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, \mathsf{true}) * \mathtt{wait}[n_1] * \mathsf{resource}) \ltimes (\mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, \mathsf{true}) * (\neg\mathtt{wait}[n_1]))
\end{aligned}$$

$$\begin{aligned}
&\mathit{SetFlag}_{\mathsf{t}}(sz) \overset{\text{def}}{=} \\
&\quad \exists tl, n_1, n_2.\ \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, \mathsf{true}) \ltimes \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, \mathsf{false})
\end{aligned}$$

$$\begin{aligned}
&\mathit{RelLock}_{\mathsf{t}}(sz) \overset{\text{def}}{=} \\
&\quad \exists tl, n_1, n_2.\ \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, \mathsf{false}) \ltimes (\mathsf{lock\_irr}_{\mathsf{t}}(tl, (n_1+1)\%sz, n_2, sz, \mathsf{true}) * \mathsf{resource})
\end{aligned}$$

$$\mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} \forall n_1.\ dp_{\mathsf{t}}(n_1) \leadsto dq_{\mathsf{t}}(n_1)$$

$$dp_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2, sz, b.\ \mathsf{lock}(\mathsf{t} :: tl, n_1, n_2, sz, b) * \mathsf{true} \wedge I$$

$$dq_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2, sz.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, (n_1+1)\%sz, n_2, sz, \mathsf{true}) * \mathsf{true} \wedge I$$

Figure 31. Invariant, precondition, rely/guarantee and definite action of counter with Anderson array-based lock.

$$\begin{aligned}&\mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, n_1, n, n_2, sz, b) \overset{\text{def}}{=}\\ &\quad \mathsf{tqueue}_{\mathsf{t}}(tl_1, tl_2, n_1, n, n_2, sz) * (\texttt{tail}\%sz = n_2) * \mathsf{fflags}(n_1, sz) * (\texttt{flag}[n_1] = b)\\ &\quad \wedge (\mathsf{list2set}(tl_1) \cap \mathsf{list2set}(\mathsf{t}::tl_2) = \emptyset) \wedge (b \vee (\neg b) \wedge (n_1 < n))\end{aligned}$$

$$\begin{aligned}&\mathsf{tqueue}_{\mathsf{t}}(tl_1, tl_2, n_1, n, n_2, sz) \overset{\text{def}}{=}\\ &\quad (0 \le n_1 \le n < n_2 < sz) \wedge \mathsf{free}(0, n_1) * \mathsf{thrds}(tl_1, n_1, n) * \mathsf{thrds}(\mathsf{t}::tl_2, n, n_2) * \mathsf{free}(n_2, sz)\\ &\quad \vee (0 \le n_2 \le n_1 \le n < sz) \wedge \exists tl_{21}, tl_{22}.\, (tl_2 = tl_{21}::tl_{22})\\ &\qquad \wedge \mathsf{thrds}(tl_{22}, 0, n_2) * \mathsf{free}(n_2, n_1) * \mathsf{thrds}(tl_1, n_1, n) * \mathsf{thrds}(\mathsf{t}::tl_{21}, n, sz)\\ &\quad \vee (0 \le n < n_2 \le n_1 < sz) \wedge \exists tl_{11}, tl_{12}.\, (tl_1 = tl_{11}::tl_{12})\\ &\qquad \wedge \mathsf{thrds}(tl_{12}, 0, n) * \mathsf{thrds}(\mathsf{t}::tl_2, n, n_2) * \mathsf{free}(n_2, n_1) * \mathsf{thrds}(tl_{11}, n_1, sz)\end{aligned}$$

$$P_0(n_1, n, n_2, b) \overset{\text{def}}{=} \exists tl_1.\, P_0(tl_1, n_1, n, n_2, b) \qquad P_3(n_1, n, n_2, b) \overset{\text{def}}{=} \exists \mathsf{t}', tl_1.\, P_0(\mathsf{t}'::tl_1, n_1, n, n_2, b)$$

$$\begin{aligned}&P_0(tl_1, n_1, n, n_2, b) \overset{\text{def}}{=}\\ &\quad \exists tl_2, sz.\, (\texttt{TNUM} = sz) * \mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, n_1, n, n_2, sz, b)\\ &\quad * (\texttt{wait}[n_1] * \mathsf{resource} \vee (\neg\texttt{wait}[n_1]) \wedge (n_1 \ne n)) * \mathsf{waits}(n_1, n_2, sz) \wedge (1 \le \mathsf{t} \le sz)\end{aligned}$$

$$P_1(n_1, n_2, b) \overset{\text{def}}{=} \exists sz, tl.\, (\texttt{TNUM} = sz) * \mathsf{lock}(\mathsf{t}::tl, n_1, n_2, sz, b) * \texttt{wait}[n_1] * \mathsf{resource} * \mathsf{waits}(n_1, n_2, sz) \wedge (1 \le \mathsf{t} \le sz)$$

$$P_2(n_1, n_2, b) \overset{\text{def}}{=} \exists sz, tl.\, (\texttt{TNUM} = sz) * \mathsf{lock}(\mathsf{t}::tl, n_1, n_2, sz, b) * (\neg\texttt{wait}[n_1]) * \mathsf{waits}(n_1, n_2, sz) \wedge (1 \le \mathsf{t} \le sz)$$

$$\mathsf{bef}(\mathsf{b}) \overset{\text{def}}{=} 0 \text{ if } \mathsf{b} = \mathsf{true} \qquad \mathsf{bef}(\mathsf{b}) \overset{\text{def}}{=} 1 \text{ if } \mathsf{b} = \mathsf{false}$$

```
inc():
 1  local i, b, r;
    { $P \wedge \mathsf{arem}(\texttt{INC})$ }
 2  < i := getAndInc(tail) % TNUM; queue[i] := cid; wait[i] := true; >
    { $\exists n_1, n, n_2, b.\ P_0(n_1, n, n_2, b) \wedge (\texttt{i} = n) \wedge \mathsf{arem}(\texttt{INC})$ }
 3  b := flag[i];
    { $\exists n_1, n, n_2, b.\ P_0(n_1, n, n_2, b) \wedge (\texttt{i} = n) \wedge (\texttt{b} \wedge (\texttt{b} = b) \wedge (n_1 = n) \vee (\neg\texttt{b})) \wedge \mathsf{arem}(\texttt{INC})$ }
    { $\exists n_1, n_2, b.\ P_0(n_1, n, n_2, b) \wedge (\texttt{i} = n) \wedge (\texttt{b} \wedge (\texttt{b} = b) \wedge (n_1 = n) \vee (\neg\texttt{b})) \wedge \mathsf{arem}(\texttt{INC}) \wedge \Diamond(\mathsf{bef}(\texttt{b}))$ }
 4  while (!b) {
 5    b := flag[i];
 6  }
    { $\exists n_1, n_2.\ P_1(n_1, n_2, \mathsf{true}) \wedge (\texttt{i} = n_1) \wedge \mathsf{arem}(\texttt{INC})$ }
 7   <wait[i] := false>;
    { $\mathsf{resource} * (\exists n_1, n_2.\ P_2(n_1, n_2, \mathsf{true}) \wedge (\texttt{i} = n_1)) \wedge \mathsf{arem}(\texttt{INC})$ }
 8  r := x; x := r + 1;
    { $(\texttt{x} = \texttt{X} + 1) * (\exists n_1, n_2.\ P_2(n_1, n_2, \mathsf{true}) \wedge (\texttt{i} = n_1)) \wedge \mathsf{arem}(\texttt{INC})$ }
 9  flag[i] := false;
    { $(\texttt{x} = \texttt{X} + 1) * (\exists n_1, n_2.\ P_2(n_1, n_2, \mathsf{false}) \wedge (\texttt{i} = n_1)) \wedge \mathsf{arem}(\texttt{INC})$ }
10  <flag[(i + 1) % TNUM] := true; queue[i] := -1>;
    { $P \wedge \mathsf{arem}(\textbf{skip})$ }
```

Here the loop at lines 4-6 is verified in Figure 33.

Figure 32. Proof outline.

$p' \overset{\text{def}}{=} \exists n_1, n_2, b.\ (P_1(n, n_2, \mathsf{true}) \wedge \Diamond(0) \vee P_3(n_1, n, n_2, b) \wedge \Diamond(\mathsf{bef}(\mathtt{b}))) \wedge (\mathtt{i} = n) \wedge (\neg\mathtt{b}) \wedge \mathsf{arem}(\mathtt{INC})$

$J \overset{\text{def}}{=} \exists n_1, n_2, b.\ P_0(n_1, n, n_2, b)$

$Q \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D})$

$G' \overset{\text{def}}{=} [I]$

$f(\mathfrak{S}) = k$ iff $\exists tl_1, n_1, n_2, b.\ (\mathfrak{S} \models P_0(tl_1, n_1, n, n_2, b)) \wedge (k = \mathsf{len}(tl_1))$

$\mathsf{len}(\epsilon) \overset{\text{def}}{=} 0 \qquad \mathsf{len}(\mathtt{t} :: tl) \overset{\text{def}}{=} 1 + \mathsf{len}(tl)$

$(\exists n_1, n_2, b.\ P_0(n_1, n, n_2, b) \wedge (\mathtt{i} = n) \wedge (\neg\mathtt{b}) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bef}(\mathtt{b})) \wedge Q * \mathsf{true})$
$\Longrightarrow (\exists n_2.\ P_1(n, n_2, \mathsf{true}) \wedge (\mathtt{i} = n) \wedge (\neg\mathtt{b}) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(0)) * (\Diamond(1) \wedge \mathsf{emp})$

$\{ p' \}$
$\{ \exists n_1, n_2, b.\ (P_1(n, n_2, \mathsf{true}) \wedge \Diamond(0) \vee P_3(n_1, n, n_2, b) \wedge \Diamond(\mathsf{bef}(\mathtt{b}))) \wedge (\mathtt{i} = n) \wedge (\neg\mathtt{b}) \wedge \mathsf{arem}(\mathtt{INC}) \}$
```
5    b := flag[i];
```
$\{ \exists n_1, n_2, b.\ P_0(n_1, n, n_2, b) \wedge (\mathtt{i} = n) \wedge (\mathtt{b} \wedge (\mathtt{b} = b) \wedge (n_1 = n) \vee (\neg\mathtt{b})) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bef}(\mathtt{b})) \}$
$\{ p \}$

We can prove:

(1) $J \Rightarrow I$; $Q \Rightarrow I$; $\mathsf{Sta}(J, G' \vee R)$.

(2) $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ (where picking $\mathcal{D}'$ used in the WHL rule as $\mathcal{D}$).

Figure 33. Proof outline – the loop at lines 4-6.

```
initialize(){
   tail := new Node(false, 0);
   tq := ϵ; acquired := false;
}

thread_initialize(){ // mynode: each thread's local variable (pointing to a shared node)
   mynode := new Node(false, cid); // fields: succwait, tid
}

inc(){
 1  local p, b, r;
 2  mynode.succwait := true;
 3  <p := getAndSet(&tail, mynode); tq := tq::cid>;
 4  b := p.succwait;
 5  while (b) {
 6    b := p.succwait;
 7  }
 8  <acquired := true>;
 9  r := x; x := r + 1;
10  <mynode.succwait := false; tq := getTail(tq); acquired := false;
     p.tid := cid; mynode.tid := 0>;
11  mynode := p;
}
```

Figure 34. Counter with CLH lock.

C.3 Counter with CLH list-based queue lock

In this section, we verify the counter implementation with the CLH list-based queue lock is linearizable with respect to the atomic operation `INC` and is starvation-free. Fig. 34 shows the concrete implementation with auxiliary code (in red). The CLH lock algorithm uses a virtual linked list, with the `tail` pointer pointing to the end of the list. Each list node contains a field `succwait`. If the field is `true`, then the corresponding thread has either acquired the lock, or is waiting for the lock (thus its successor thread must be waiting for the lock). If the field is `false`, then the thread has released the lock (i.e., its successor thread acquires the lock and is no longer waiting). We say the list is "virtual" because there is no explicit "next" pointer in each node pointing to the successor node, and each thread refers to its predecessor through a thread-local pointer `p` in the code.

Initially the list contains only the `tail` node, and each thread has its own node `mynode` (here `mynode` is a thread-local pointer, but the node is shared, since the successor thread may keep a reference to the node). To acquire the lock, the thread first sets its `succwait` field to `true` to indicate that the thread is requesting the lock (line `2`). Next it makes its own node the tail of the list, simultaneously getting a reference `p` to its predecessor, i.e., the original tail node (line `3`). Then the thread spins on the predecessor's `succwait` field until the predecessor releases the lock (lines `4-7`). To release the lock, the thread sets its `succwait` field to `false` (line `10`). It reuses its predecessor's node `p` as its new own node for future lock accesses (line `11`).

To verify the code, we give each list node an auxiliary field `tid` to record the thread which owns the node. If no thread owns the node, the auxiliary `tid` field is 0. We also introduce an auxiliary variable `tq` which is a mathematical list with "::" for concatenation. `tq` plays a similar role as the auxiliary array `queue` in the previous proofs of Anderson array-based lock. It records the queue of the threads requesting the lock. The auxiliary variable `acquired` plays a similar role as the auxiliary array `wait` in the previous proofs of Anderson array-based lock, but `acquired` is not a per-thread flag. It is set to `true` when the lock is acquired by some thread and reset to `false` when the lock is released.

In the code, we append the current thread ID `cid` at the end of the thread queue `tq` at the same time when the thread makes its own node the tail of the list (line `3`). The auxiliary variable `acquired` is set to `true` after the spinning loop (line `8`). We view line `10` as the time of releasing the lock, so we remove the head thread in `tq` and reset `acquired` at the same time. Simultaneously we set the predecessor node `p`'s `tid` field to `cid` and the currently-owned node's `tid` field to 0. That is, when releasing the lock, the current thread will reuse the predecessor's node and its original node is no longer owned by any thread. Line `11` physically sets the thread-local pointer `mynode`, but actually at line `10` of releasing the lock, the thread already switches to own the predecessor node.

Fig. 35 shows the full definitions of the precise invariant, the precondition, the rely and guarantee conditions and the definite action. The definitions are similar the the corresponding ones for Anderson array-based lock. Since the list is virtual, we introduce two mappings *ta* and *tb* to record the address and the boolean `succwait` field of each thread's node. Then as shown in the definition of $\mathsf{queue}(tl, x, ta, tb)$ (which is the key to define the invariant I), the thread queue is recorded in `tq` and the nodes of these threads constitute a virtual list $\mathsf{seg}(tl, x, z, ta, tb)$. We also have an additional node $\mathsf{node}(x, \mathsf{false}, 0)$ as the predecessor of the virtual list. If the virtual list is empty, then `tail` points to this additional node x; otherwise `tail` points to the end node of the virtual list. The first thread `t` at the virtual list acquires the lock. Its `succwait` field must be `true`, indicating that its successor thread must be waiting now. Its predecessor x's `succwait` field is false, saying that `t` is no longer waiting and has acquired the lock.

Fig. 36 shows the proof outline, which is similar to the previous proofs of Anderson array-based lock.

$tl ::= \epsilon \mid \mathsf{t} :: tl \qquad ta \in ThrdID \rightharpoonup Nat \qquad tb \in ThrdID \rightharpoonup Bool$

$\mathsf{list2set}(\epsilon) \stackrel{\text{def}}{=} \emptyset$

$\mathsf{list2set}(\mathsf{t} :: tl) \stackrel{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(tl)$

$I \stackrel{\text{def}}{=} \exists sz, tl, x_0, ta, tb.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(tl, x_0, ta, tb, sz) * ((\neg\mathtt{acquired}) * \mathsf{resource} \vee \mathtt{acquired} \wedge (tl \neq \epsilon))$

$\mathsf{resource} \stackrel{\text{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad \mathsf{lres} \stackrel{\text{def}}{=} (\mathtt{x} = \_)$

$\mathsf{lock}(tl, x_0, ta, tb, sz) \stackrel{\text{def}}{=}$
$\quad (dom(ta) = dom(tb) = \{1, \ldots, sz\})$
$\quad \wedge\ \mathsf{queue}(tl, x_0, ta, tb) * \mathsf{nodeset}(\{\mathsf{t} \mid (1 \leq \mathsf{t} \leq sz) \wedge (\mathsf{t} \notin \mathsf{list2set}(tl))\}, ta, tb)$

$\mathsf{lock\_irr}_{\mathsf{t}}(tl, x_0, ta, tb, sz) \stackrel{\text{def}}{=} (\mathsf{t} \notin \mathsf{list2set}(tl)) \wedge \mathsf{lock}(tl, x_0, ta, tb, sz)$

$\mathsf{queue}(tl, x, ta, tb) \stackrel{\text{def}}{=} \exists z.\ (\mathtt{tq} = tl) * \mathsf{node}(x, \mathsf{false}, 0) * \mathsf{seg}(tl, x, z, ta, tb) * (\mathtt{tail} = z)$

$\mathsf{seg}(tl, x, z, ta, tb) \stackrel{\text{def}}{=}$
$\quad (tl = \epsilon) \wedge \mathsf{emp} \wedge (x = z)$
$\quad \vee\ \exists \mathsf{t}, tl', y.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (ta(\mathsf{t}) = y) \wedge (tb(\mathsf{t}) = \mathsf{true}) \wedge \mathsf{node}(y, \mathsf{true}, \mathsf{t}) * \mathsf{seg}(tl', y, z, ta, tb)$

$\mathsf{nodeset}(S, ta, tb) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t} \in S} \mathsf{node}(ta(\mathsf{t}), tb(\mathsf{t}), \mathsf{t}))$

$\mathsf{node}(x, b, s) \stackrel{\text{def}}{=} (x.\mathtt{succwait} = b) * (x.\mathtt{tid} = s)$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} P_{\mathsf{t}}(\mathtt{mynode}, \mathsf{false})$

$P_{\mathsf{t}}(x, b) \stackrel{\text{def}}{=}$
$\quad \exists x_0, ta, tb, sz, tl.\ (\mathtt{TNUM} = sz) * \mathsf{lock\_irr}_{\mathsf{t}}(tl, x_0, ta \uplus \{\mathsf{t} \leadsto x\}, tb \uplus \{\mathsf{t} \leadsto b\}, sz)$
$\quad * ((\neg\mathtt{acquired}) * \mathsf{resource} \vee \mathtt{acquired} \wedge (tl \neq \epsilon)) \wedge (1 \leq \mathsf{t} \leq sz)$

$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists sz.\ (\mathit{SetSuccwait}_{\mathsf{t}}(sz) \vee \mathit{ReqLock}_{\mathsf{t}}(sz) \vee \mathit{AcqResource}_{\mathsf{t}}(sz) \vee \mathit{RelLock}_{\mathsf{t}}(sz) \vee \mathsf{Id})$
$\quad * [(\mathtt{TNUM} = sz) \wedge (1 \leq \mathsf{t} \leq sz)] * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{SetSuccwait}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=}$
$\quad \exists tl, x_0, ta, tb.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, x_0, ta, tb \uplus \{\mathsf{t} \leadsto \mathsf{false}\}, sz) \ltimes \mathsf{lock\_irr}_{\mathsf{t}}(tl, x_0, ta, tb \uplus \{\mathsf{t} \leadsto \mathsf{true}\}, sz)$

$\mathit{ReqLock}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=}$
$\quad \exists tl, x_0, ta, tb.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, x_0, ta, tb, sz) \ltimes \mathsf{lock}(tl :: \mathsf{t}, x_0, ta, tb, sz)$

$\mathit{AcqResource}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=}$
$\quad \exists tl, x_0, ta, tb.\ (\mathsf{lock}(\mathsf{t} :: tl, x_0, ta, tb, sz) * (\neg\mathtt{acquired}) * \mathsf{resource}) \ltimes (\mathsf{lock}(\mathsf{t} :: tl, x_0, ta, tb, sz) * \mathtt{acquired})$

$\mathit{RelLock}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=}$
$\quad \exists tl, x_0, ta, tb.\ (\mathsf{lock}(\mathsf{t} :: tl, x_0, ta, tb, sz) * \mathtt{acquired})$
$\quad \ltimes (\mathsf{lock\_irr}_{\mathsf{t}}(tl, ta(\mathsf{t}), ta\{\mathsf{t} \leadsto x_0\}, tb\{\mathsf{t} \leadsto \mathsf{false}\}, sz) * (\neg\mathtt{acquired}) * \mathsf{resource})$

$\mathcal{D}_{\mathsf{t}} \stackrel{\text{def}}{=} \forall x_0, ta.\ dp_{\mathsf{t}}(x_0, ta) \leadsto dq_{\mathsf{t}}(x_0, ta)$

$dp_{\mathsf{t}}(x_0, ta) \stackrel{\text{def}}{=} \exists tl, tb, sz.\ \mathsf{lock}(\mathsf{t} :: tl, x_0, ta, tb, sz) * \mathsf{true} \wedge I$

$dq_{\mathsf{t}}(x_0, ta) \stackrel{\text{def}}{=} \exists tl, tb, sz.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, ta(\mathsf{t}), ta\{\mathsf{t} \leadsto x_0\}, tb\{\mathsf{t} \leadsto \mathsf{false}\}, sz) * \mathsf{true} \wedge I$

Figure 35. Invariant, precondition, rely/guarantee and definite action.

$\mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, x_0, p, ta, tb, sz) \overset{\text{def}}{=}$
$\quad (dom(ta) = dom(tb) = \{1, \ldots, sz\}) \land (\mathsf{list2set}(tl_1) \cap \mathsf{list2set}(\mathsf{t} :: tl_2) = \emptyset)$
$\quad \land \mathsf{tqueue}_{\mathsf{t}}(tl_1, tl_2, x_0, p, ta, tb) * \mathsf{nodeset}(\{\mathsf{t}' \mid (1 \leq \mathsf{t}' \leq sz) \land (\mathsf{t}' \notin \mathsf{list2set}(tl_1 :: \mathsf{t} :: tl_2))\}, ta, tb)$

$\mathsf{tqueue}_{\mathsf{t}}(tl_1, tl_2, x_0, p, ta, tb) \overset{\text{def}}{=}$
$\quad \exists z.\ (\mathtt{tq} = tl_1 :: \mathsf{t} :: tl_2) * \mathsf{node}(x_0, \mathsf{false}, 0) * \mathsf{seg}(tl_1, x_0, p, ta, tb) * \mathsf{seg}(\mathsf{t} :: tl_2, p, z, ta, tb) * (\mathtt{tail} = z)$

$P_0(x_0, p, x, b) \overset{\text{def}}{=} \exists tl_1.\ P_0(tl_1, x_0, p, x, b)$ $\qquad P_3(x_0, p, x, b) \overset{\text{def}}{=} \exists \mathsf{t}', tl_1.\ P_0(\mathsf{t}' :: tl_1, x_0, p, x, b)$

$P_0(tl_1, x_0, p, x, b) \overset{\text{def}}{=}$
$\quad \exists tl_2, sz, ta, tb.\ (\mathtt{TNUM} = sz) * \mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, x_0, p, ta \uplus \{\mathsf{t} \leadsto x\}, tb \uplus \{\mathsf{t} \leadsto b\}, sz)$
$\quad * ((\neg \mathtt{acquired}) * \mathsf{resource} \lor \mathtt{acquired} \land (tl_1 \neq \epsilon)) \land (1 \leq \mathsf{t} \leq sz)$

$P_1(p, x, b) \overset{\text{def}}{=}$
$\quad \exists sz, tl, ta, tb.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(\mathsf{t} :: tl, p, ta \uplus \{\mathsf{t} \leadsto x\}, tb \uplus \{\mathsf{t} \leadsto b\}, sz) * (\neg \mathtt{acquired}) * \mathsf{resource} \land (1 \leq \mathsf{t} \leq sz)$

$P_2(p, x, b) \overset{\text{def}}{=}$
$\quad \exists sz, tl, ta, tb.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(\mathsf{t} :: tl, p, ta \uplus \{\mathsf{t} \leadsto x\}, tb \uplus \{\mathsf{t} \leadsto b\}, sz) * \mathtt{acquired} \land (1 \leq \mathsf{t} \leq sz)$

$\mathsf{bet}(\mathtt{b}) \overset{\text{def}}{=} 1 \text{ if } \mathtt{b} = \mathsf{true}$ $\qquad \mathsf{bet}(\mathtt{b}) \overset{\text{def}}{=} 0 \text{ if } \mathtt{b} = \mathsf{false}$

```
inc():
 1  local p, b, r;
    { $P \land \mathsf{arem}(\mathtt{INC})$ }
 2  mynode.succwait := true;
    { $P(\mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC})$ }
 3  < p := getAndSet(&tail, mynode); tq := tq::cid>;
    { $\exists x_0.\ P_0(x_0, \mathtt{p}, \mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC})$ }
 4  b := p.succwait;
    { $\exists x_0.\ P_0(x_0, \mathtt{p}, \mathtt{mynode}, \mathsf{true}) \land ((\neg \mathtt{b}) \land (\mathtt{p} = x_0) \lor \mathtt{b}) \land \mathsf{arem}(\mathtt{INC})$ }
    { $\exists x_0.\ (\mathtt{p} = p) \land (\mathtt{mynode} = x) \land P_0(x_0, p, x, \mathsf{true}) \land ((\neg \mathtt{b}) \land (p = x_0) \lor \mathtt{b}) \land \mathsf{arem}(\mathtt{INC}) \land \Diamond(\mathsf{bet}(\mathtt{b}))$ }
 5  while (b) {
 6    b := p.succwait;
 7  }
    { $P_1(\mathtt{p}, \mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC})$ }
 8   <acquired := true>;
    { $\mathsf{resource} * P_2(\mathtt{p}, \mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC})$ }
 9  r := x; x := r + 1;
    { $(\mathtt{x} = \mathtt{X} + 1) * P_2(\mathtt{p}, \mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC})$ }
10  < mynode.succwait := false; tq := getTail(tq); acquired := false;
      p.tid := cid; mynode.tid := 0>;
    { $(\mathtt{mynode} = \_) \land P(\mathtt{p}, \mathsf{false}) \land \mathsf{arem}(\mathbf{skip})$ }
11  mynode := p;
    { $P \land \mathsf{arem}(\mathbf{skip})$ }
```

Here the loop at lines 5-7 is verified in Figure 37.

Figure 36. Proof outline.

$p' \overset{\text{def}}{=} \exists x_0.\, (P_1(p, x, \mathsf{true}) \wedge \Diamond(0) \vee P_3(x_0, p, x, \mathsf{true}) \wedge \Diamond(\mathsf{bet}(\mathtt{b}))) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC})$

$J \overset{\text{def}}{=} \exists x_0.\, P_0(x_0, p, x, \mathsf{true})$

$Q \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D})$

$G' \overset{\text{def}}{=} [I]$

$f(\mathfrak{S}) = k$ iff $\exists tl_1, x_0.\, (\mathfrak{S} \models P_0(tl_1, x_0, p, x, \mathsf{true})) \wedge (k = \mathsf{len}(tl_1))$

$\mathsf{len}(\epsilon) \overset{\text{def}}{=} 0 \qquad \mathsf{len}(\mathtt{t} :: tl) \overset{\text{def}}{=} 1 + \mathsf{len}(tl)$

$$(\exists x_0.\, (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge P_0(x_0, p, x, \mathsf{true}) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bet}(\mathtt{b})) \wedge Q * \mathsf{true})$$
$$\Longrightarrow ((\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge P_1(p, x, \mathsf{true}) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(0)) * (\Diamond(1) \wedge \mathsf{emp})$$

$\{ p' \}$
$\{ \exists x_0.\, (P_1(p, x, \mathsf{true}) \wedge \Diamond(0) \vee P_3(x_0, p, x, \mathsf{true}) \wedge \Diamond(\mathsf{bet}(\mathtt{b}))) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC}) \}$

```
6    b := p.succwait;
```

$\{ \exists x_0.\, P_0(x_0, p, x, \mathsf{true}) \wedge ((\neg \mathtt{b}) \wedge (p = x_0) \vee \mathtt{b}) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bet}(\mathtt{b})) \}$

We can prove:

(1) $J \Rightarrow I; Q \Rightarrow I; \mathsf{Sta}(J, G' \vee R)$.

(2) $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ (where picking $\mathcal{D}'$ needed in the WHL rule as $\mathcal{D}$).

Figure 37. Proof outline – the loop at lines 5-7.

```
initialize(){
   tail := null; tq := ϵ; acquired := false;
}

thread_initialize(){ // mynode: each thread's local variable (pointing to a shared node)
   mynode := new Node(false, null, cid); // fields: locked, next, tid
}

inc(){
 1  local p, s, b, nos, r;
 2  <p := getAndSet(&tail, mynode); tq := tq::cid>;
 3  if (p != null) {
 4    mynode.locked := true;
 5    p.next := mynode;
 6    b := mynode.locked;
 7    while (b) {
 8      b := mynode.locked;
 9    }
10  }
11  <acquired := true>;
12  r := x; x := r + 1;
13  s := mynode.next;
14  if (s = null) {
15    <nos := cas(&tail, mynode, null); if(nos) { tq := getTail(tq); acquired := false; } >;
16    if (!nos) {
17      s := mynode.next;
18      while (s = null) {
19        s := mynode.next;
20      }
21    }
22  }
23  if (s != null) {
24    <s.locked := false; tq := getTail(tq); acquired := false>;
25    mynode.next := null;
26  }
}
```

Figure 38. Counter with MCS lock.

C.4 Counter with MCS list-based queue lock

In this section, we verify the counter implementation with the MCS list-based queue lock is linearizable with respect to the atomic operation `INC` and is starvation-free. Fig. 38 shows the concrete implementation with auxiliary code (in red). Unlike the CLH lock algorithm, the MCS lock algorithm uses an *explicit* linked list, with `tail` pointing to the end of the list. Each list node contains a `locked` field, indicating whether the corresponding thread is waiting for the lock, and a `next` field, pointing to the next node in the list.

Initially the list is empty and `tail` points to `null`. Each thread owns a node `mynode`. To acquire the lock, a thread appends its own node at the tail of the list (line `2`). If the queue is not empty originally, then the thread sets its predecessor node's `next` field to refer to its own node (line `5`). The thread then spins on the `locked` field of its own node, waiting until its predecessor sets this field to false (lines `6-9`).

To release the lock, the thread checks whether it has a successor (lines `13-22`). It first checks if its node's `next` field is `null` (line `14`). If so, then either no other thread is requesting the lock, or there is another thread but it is slow. To distinguish the two cases, the current thread checks if its own node is at the `tail` of the list, and set `tail` to `null` if it is the case (line `15`). Otherwise, its own node is not at the tail, thus we know some successor thread is requesting the lock. Then the current thread spins until the successor thread links its node as the next node of the current thread (lines `17-20`). When the successor node appears, the current thread sets its successor's `locked` field to false (line `24`), indicating that the successor has gets the lock and does not need to wait anymore.

To verify the code, we introduce several auxiliary structures following the proofs of the CLH lock. we give each list node an auxiliary field `tid` to record the thread which owns the node. Unlike the CLH lock algorithm, here a thread does not change its own node, thus this `tid` field is set at the beginning and no longer modified during the executions. We also introduce the auxiliary variable `tq` to record the list of the threads requesting the lock. The auxiliary variable `acquired` indicates whether the lock has been acquired by some thread.

In the code, we append the current thread ID `cid` at the end of the thread queue `tq` at the same time when the thread makes its own node at the tail of the list (line `2`). The auxiliary variable `acquired` is set to `true` after the spinning loop (line `11`). When we are sure that there is no successor thread (i.e., the `cas` succeeds at line `15`), the current thread releases the lock by removing the head thread in `tq` and resetting `acquired` at the same time. If there exists some successor thread, the lock is released at line `24`.

Fig. 39 defines the precise invariant and the precondition. The lock satisfies either unlocked or locked. For the case of locked, we know the threads requesting the lock constitute the queue `tq`. The head thread in `tq` acquires the lock. Its `locked` bit must be false since it does not need to wait for the lock. Each remaining thread in `tq` is waiting, so either its `locked` bit is `true` and the thread has set its predecessor node's `next` field to refer to its own node, or we record the corresponding thread in a set S (the thread is slow and has not linked its node as the next of its predecessor), as defined in the predicate lls. For all the other threads which do not request the lock, we use nodeset for their nodes in

$tl ::= \epsilon \mid \mathsf{t}::tl \qquad ta \in \mathit{ThrdID} \rightharpoonup \mathit{Nat} \qquad tb \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$\mathsf{list2set}(\epsilon) \stackrel{\mathrm{def}}{=} \emptyset \qquad \mathsf{list2set}(\mathsf{t}::tl) \stackrel{\mathrm{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(tl)$

$I \stackrel{\mathrm{def}}{=} \exists sz, tl, ta, tb, S.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(tl, ta, tb, sz, S) * ((\neg\mathtt{acquired}) * \mathsf{resource} \vee \mathtt{acquired} \wedge (tl \neq \epsilon))$

$\mathsf{resource} \stackrel{\mathrm{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad \mathsf{lres} \stackrel{\mathrm{def}}{=} (\mathtt{x} = \_)$

$\mathsf{lock}(tl, ta, tb, sz, S) \stackrel{\mathrm{def}}{=} \exists y.\ \mathsf{lock}(tl, ta, tb, y, sz, S)$

$\mathsf{lock}(tl, ta, tb, y, sz, S) \stackrel{\mathrm{def}}{=}$
$\quad (dom(ta) = dom(tb) = \{1, \ldots, sz\})$
$\quad \wedge (\mathsf{unlocked}(tl) \wedge (S = \emptyset) \vee \mathsf{locked}(tl, ta, tb, y, S)) * \mathsf{nodeset}(\{\mathsf{t} \mid (1 \leq \mathsf{t} \leq sz) \wedge (\mathsf{t} \notin \mathsf{list2set}(tl))\}, ta, tb)$

$\mathsf{lock\_irr}_{\mathsf{t}}(tl, ta, tb, y, sz, S) \stackrel{\mathrm{def}}{=}$
$\quad \exists x.\ (\mathsf{t} \notin \mathsf{list2set}(tl)) \wedge (dom(ta) = dom(tb) = \{1, \ldots, sz\}) \wedge (ta(\mathsf{t}) = x) \wedge (tb(\mathsf{t}) = \mathsf{false})$
$\quad \wedge (\mathsf{unlocked}(tl) \vee \mathsf{locked}(tl, ta, tb, S)) * \mathsf{node}_{\mathsf{t}}(x, \mathsf{false}, y) * \mathsf{nodeset}(\{\mathsf{t}' \mid (1 \leq \mathsf{t}' \leq sz) \wedge (\mathsf{t}' \neq \mathsf{t}) \wedge (\mathsf{t}' \notin \mathsf{list2set}(tl))\}, ta, tb)$

$\mathsf{unlocked}(tl) \stackrel{\mathrm{def}}{=} (\mathtt{tq} = tl) * (\mathtt{tail} = \mathtt{null}) \wedge (tl = \epsilon) \qquad \mathsf{locked}(tl, ta, tb, S) \stackrel{\mathrm{def}}{=} \exists y.\ \mathsf{locked}(tl, ta, tb, y, S)$

$\mathsf{locked}(tl, ta, tb, y, S) \stackrel{\mathrm{def}}{=}$
$\quad \exists \mathsf{t}, tl', x, z.\ (tl = \mathsf{t}::tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (ta(\mathsf{t}) = x) \wedge (tb(\mathsf{t}) = \mathsf{false})$
$\quad \wedge (\mathtt{tq} = tl) * \mathsf{node}_{\mathsf{t}}(x, \mathsf{false}, y) * \mathsf{lls}(tl', x, y, z, \mathtt{null}, ta, tb, S) * (\mathtt{tail} = z)$

$\mathsf{lls}(tl, p, x, z, n, ta, tb, S) \stackrel{\mathrm{def}}{=}$
$\quad (tl = \epsilon) \wedge (p = z) \wedge (x = n) \wedge (S = \emptyset)$
$\quad \vee \exists \mathsf{t}, tl', x', b, S'.\ (tl = \mathsf{t}::tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (ta(\mathsf{t}) = x') \wedge (tb(\mathsf{t}) = b)$
$\quad\quad \wedge \mathsf{node}_{\mathsf{t}}(x', b, y) * \mathsf{lls}(tl', x', y, z, n, ta, tb, S') \wedge ((x = x') \wedge (b = \mathsf{true}) \wedge (S = S') \vee (x = \mathtt{null}) \wedge (S = \{\mathsf{t}\} \uplus S'))$

$\mathsf{nodeset}(S, ta, tb) \stackrel{\mathrm{def}}{=} (\circledast_{\mathsf{t} \in S}(\mathsf{node}_{\mathsf{t}}(ta(\mathsf{t}), \mathsf{false}, \_) \wedge (tb(\mathsf{t}) = \mathsf{false})))$

$\mathsf{node}_s(x, b, y) \stackrel{\mathrm{def}}{=} (x.\mathtt{locked} = b) * (x.\mathtt{next} = y) * (x.\mathtt{tid} = s)$

$P_{\mathsf{t}} \stackrel{\mathrm{def}}{=} P_{\mathsf{t}}(\mathtt{mynode}, \mathtt{null})$

$P_{\mathsf{t}}(x, y) \stackrel{\mathrm{def}}{=}$
$\quad \exists ta, tb, sz, tl, S.\ (\mathtt{TNUM} = sz) * \mathsf{lock\_irr}_{\mathsf{t}}(tl, ta \uplus \{\mathsf{t} \leadsto x\}, tb, y, sz, S)$
$\quad * ((\neg\mathtt{acquired}) * \mathsf{resource} \vee \mathtt{acquired} \wedge (tl \neq \epsilon)) \wedge (1 \leq \mathsf{t} \leq sz)$

Figure 39. Invariant and precondition.

the shared state. Their `locked` bits are all false. As in the proof of CLH lock, we use two mappings *ta* and *tb* to record the address and the boolean `locked` field of each thread's node.

Fig. 40 defines the rely and guarantee conditions and the definite actions. We have several actions since the code is very fine-grained. The action *ReqLock* is for line 2, where the thread makes its node the tail of the list. If it has a predecessor, then *SetLocked* sets its `locked` bit (line 4), and *SetPred* sets its predecessor node's `next` to its own node (line 5). The head thread in `tq` does *AcqResource* (line 11) to set `acquired` and transfer the shared resource. To release the lock, we distinguish the case when there is no successor thread (*RelLock1*, corresponding to line 15) and the opposite case (*RelLock2*, corresponding to line 24). Finally, the thread does *ResetNext* to set the `next` field of its own node to `null`.

Note here we have *two* definite actions. $dp1 \leadsto dq1$ corresponds to the action *RelLock2*, which says the thread holding the lock will finally release the lock by setting the successor thread's `locked` bit to false. The other definite action $dp2 \leadsto dq2$ corresponds to *SetPred*, saying that the current slow thread which has requested the lock will finally link its node in the list by setting its predecessor node's `next` field to refer to its own node.

Fig. 41 and Fig. 43 show the proof outline. The proofs are similar to the previous proofs of Anderson array-based lock. Note that the two loops in the code makes use of the two definite actions respectively.

$$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$\begin{aligned}
G_{\mathsf{t}} \stackrel{\text{def}}{=}\ & \\
& \exists sz.\ (\mathit{ReqLock}_{\mathsf{t}}(sz) \vee \mathit{SetLocked}_{\mathsf{t}}(sz) \vee \mathit{SetPred}_{\mathsf{t}}(sz) \vee \mathit{AcqResource}_{\mathsf{t}}(sz) \\
& \qquad \vee \mathit{RelLock1}_{\mathsf{t}}(sz) \vee \mathit{RelLock2}_{\mathsf{t}}(sz) \vee \mathit{ResetNext}_{\mathsf{t}}(sz) \vee \mathsf{Id}) \\
& * [(\mathtt{TNUM} = sz) \wedge (1 \leq \mathsf{t} \leq sz)] * \mathsf{Id} \wedge (I \ltimes I)
\end{aligned}$$

$$\begin{aligned}
& \mathit{ReqLock}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists tl, ta, tb, S.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, ta, tb, \mathtt{null}, sz, S) \ltimes \mathsf{lock}(tl :: \mathsf{t}, ta, tb, sz, S \uplus \{\mathsf{t}\})
\end{aligned}$$

$$\begin{aligned}
& \mathit{SetLocked}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists tl_1, tl_2, ta, tb, S.\ ((tl \neq \epsilon) \wedge \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2, ta, tb \uplus \{\mathsf{t} \rightsquigarrow \mathsf{false}\}, sz, S)) \ltimes \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2, ta, tb \uplus \{\mathsf{t} \rightsquigarrow \mathsf{true}\}, sz, S)
\end{aligned}$$

$$\begin{aligned}
& \mathit{SetPred}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists tl_1, tl_2, ta, tb, S.\ \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2, ta, tb \uplus \{\mathsf{t} \rightsquigarrow \mathsf{true}\}, sz, S \uplus \{\mathsf{t}\}) \ltimes \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2, ta, tb \uplus \{\mathsf{t} \rightsquigarrow \mathsf{true}\}, sz, S)
\end{aligned}$$

$$\begin{aligned}
& \mathit{AcqResource}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists tl, ta, tb, S.\ (\mathsf{lock}(\mathsf{t} :: tl, ta, tb, sz, S) * (\neg\mathtt{acquired}) * \mathsf{resource}) \ltimes (\mathsf{lock}(\mathsf{t} :: tl, ta, tb, sz, S) * \mathtt{acquired})
\end{aligned}$$

$$\begin{aligned}
& \mathit{RelLock1}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists ta, tb.\ (\mathsf{lock}(\mathsf{t}, ta, tb, sz, \emptyset) * \mathtt{acquired}) \ltimes (\mathsf{lock\_irr}_{\mathsf{t}}(\epsilon, ta, tb, \mathtt{null}, sz, \emptyset) * (\neg\mathtt{acquired}) * \mathsf{resource})
\end{aligned}$$

$$\begin{aligned}
& \mathit{RelLock2}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists \mathsf{t}', tl, ta, tb, S.\ (\mathsf{lock}(\mathsf{t} :: \mathsf{t}' :: tl, ta, tb, sz, S) * \mathtt{acquired} \wedge (\mathsf{t}' \notin S)) \\
& \quad \ltimes (\mathsf{lock\_irr}_{\mathsf{t}}(\mathsf{t}' :: tl, ta, tb\{\mathsf{t}' \rightsquigarrow \mathsf{false}\}, ta(\mathsf{t}'), sz, S) * (\neg\mathtt{acquired}) * \mathsf{resource})
\end{aligned}$$

$$\begin{aligned}
& \mathit{ResetNext}_{\mathsf{t}}(sz) \stackrel{\text{def}}{=} \\
& \quad \exists tl, ta, tb, S.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, ta, tb, \_, sz, S) \ltimes \mathsf{lock\_irr}_{\mathsf{t}}(tl, ta, tb, \mathtt{null}, sz, S)
\end{aligned}$$

$$\mathcal{D}_{\mathsf{t}} \stackrel{\text{def}}{=} (\forall \mathsf{t}', ta, tl.\ dp1_{\mathsf{t}}(\mathsf{t}', ta, tl) \rightsquigarrow dq1_{\mathsf{t}}(\mathsf{t}', ta, tl)) \wedge (\forall tl_1, ta, tl_2.\ dp2_{\mathsf{t}}(tl_1, ta, tl_2) \rightsquigarrow dq2_{\mathsf{t}}(tl_1, ta, tl_2))$$

$$dp1_{\mathsf{t}}(\mathsf{t}', ta, tl) \stackrel{\text{def}}{=} \exists tl', tb, sz, S.\ \mathsf{lock}(\mathsf{t} :: \mathsf{t}' :: tl :: tl', ta, tb, sz, S) * \mathsf{true} \wedge (\mathsf{t}' \notin S) \wedge I$$

$$dq1_{\mathsf{t}}(\mathsf{t}', ta, tl) \stackrel{\text{def}}{=} \exists tl', tb, sz, S.\ \mathsf{lock\_irr}_{\mathsf{t}}(\mathsf{t}' :: tl :: tl', ta, tb\{\mathsf{t}' \rightsquigarrow \mathsf{false}\}, ta(\mathsf{t}'), sz, S) * \mathsf{true} \wedge I$$

$$dp2_{\mathsf{t}}(tl_1, ta, tl_2) \stackrel{\text{def}}{=} \exists tl_2', tb, sz, S.\ \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2 :: tl_2', ta, tb \uplus \{\mathsf{t} \rightsquigarrow \_\}, sz, S \uplus \{\mathsf{t}\}) * \mathsf{true} \wedge I$$

$$dq2_{\mathsf{t}}(tl_1, ta, tl_2) \stackrel{\text{def}}{=} \exists tl_2', tb, sz, S.\ \mathsf{lock}(tl_1 :: \mathsf{t} :: tl_2 :: tl_2', ta, tb \uplus \{\mathsf{t} \rightsquigarrow \mathsf{true}\}, sz, S) * \mathsf{true} \wedge I$$

Figure 40. Rely/guarantee and definite action.

$$\mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, p, x, ta, tb, sz, S_1, S_2) \overset{\text{def}}{=}$$
$$\quad (dom(ta) = dom(tb) = \{1, \ldots, sz\}) \land (\mathsf{list2set}(tl_1) \cap \mathsf{list2set}(\mathsf{t} :: tl_2) = \emptyset)$$
$$\quad \land\, \mathsf{twait}_{\mathsf{t}}(tl_1, tl_2, p, x, ta, tb, S_1, S_2) * \mathsf{nodeset}(\{\mathsf{t}' \mid (1 \le \mathsf{t}' \le sz) \land (\mathsf{t}' \notin \mathsf{list2set}(tl_1 :: \mathsf{t} :: tl_2))\}, ta, tb)$$

$$\mathsf{twait}_{\mathsf{t}}(tl_1, tl_2, p, x, ta, tb, S_1, S_2) \overset{\text{def}}{=}$$
$$\quad \exists \mathsf{t}', tl', x', y', z.\ (tl_1 = \mathsf{t}' :: tl') \land (\mathsf{t}' \notin \mathsf{list2set}(tl')) \land (ta(\mathsf{t}') = x') \land (tb(\mathsf{t}') = \mathsf{false})$$
$$\quad \land\, (\mathtt{tq} = tl_1 :: \mathsf{t} :: tl_2) * \mathsf{node}_{\mathsf{t}'}(x', \mathsf{false}, y') * \mathsf{lls}(tl', x', y', p, x, ta, tb, S_1) * \mathsf{lls}(\mathsf{t} :: tl_2, p, x, z, \mathtt{null}, ta, tb, S_2) * (\mathtt{tail} = z)$$

$$P_0(p, x, x', b) \overset{\text{def}}{=} \exists tl_1, tl_2, S_1, S_2.\ P_0(tl_1, tl_2, p, x, x', b, S_1, S_2) \qquad P_2(x, b) \overset{\text{def}}{=} \exists tl, y.\ P_2(tl, x, b, y)$$

$$P_0(tl_1, tl_2, p, x, x', b, S_1, S_2) \overset{\text{def}}{=}$$
$$\quad \exists sz, ta, tb.\ (\mathtt{TNUM} = sz) * \mathsf{tlocked}_{\mathsf{t}}(tl_1, tl_2, p, x, ta \uplus \{\mathsf{t} \rightsquigarrow x'\}, tb \uplus \{\mathsf{t} \rightsquigarrow b\}, sz, S_1, S_2)$$
$$\quad * ((\neg \mathtt{acquired}) * \mathsf{resource} \lor \mathtt{acquired}) \land (1 \le \mathsf{t} \le sz)$$

$$P_1(x, b) \overset{\text{def}}{=}$$
$$\quad \exists sz, tl, ta, tb, S.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(\mathsf{t} :: tl, ta \uplus \{\mathsf{t} \rightsquigarrow x\}, tb \uplus \{\mathsf{t} \rightsquigarrow b\}, sz, S) * (\neg \mathtt{acquired}) * \mathsf{resource} \land (1 \le \mathsf{t} \le sz)$$

$$P_2(tl, x, b, y) \overset{\text{def}}{=}$$
$$\quad \exists sz, ta, tb, S.\ (\mathtt{TNUM} = sz) * \mathsf{lock}(\mathsf{t} :: tl, ta \uplus \{\mathsf{t} \rightsquigarrow x\}, tb \uplus \{\mathsf{t} \rightsquigarrow b\}, y, sz, S) * \mathtt{acquired} \land (1 \le \mathsf{t} \le sz)$$

$\mathsf{bet}(\mathtt{b}) \overset{\text{def}}{=} 1$ if $\mathtt{b} = \mathsf{true}$ $\qquad$ $\mathsf{bet}(\mathtt{b}) \overset{\text{def}}{=} 0$ if $\mathtt{b} = \mathsf{false}$

$\mathsf{pez}(\mathtt{s}) \overset{\text{def}}{=} 1$ if $\mathtt{s} = \mathtt{null}$ $\qquad$ $\mathsf{pez}(\mathtt{s}) \overset{\text{def}}{=} 0$ if $\mathtt{s} \neq \mathtt{null}$

```
inc():
 1  local p, s, b, nos, r;
    $\{ P \land \mathsf{arem}(\mathtt{INC}) \}$
 2  < p := getAndSet(&tail, mynode); tq := tq::cid>;
    $\{ (P_0(\mathtt{p}, \mathtt{null}, \mathtt{mynode}, \mathsf{false}) \lor (\mathtt{p} = \mathtt{null}) \land P_1(\mathtt{mynode}, \mathsf{false})) \land \mathsf{arem}(\mathtt{INC}) \}$
 3  if (p != null) {
      $\{ P_0(\mathtt{p}, \mathtt{null}, \mathtt{mynode}, \mathsf{false}) \land \mathsf{arem}(\mathtt{INC}) \}$
 4    mynode.locked := true;
      $\{ P_0(\mathtt{p}, \mathtt{null}, \mathtt{mynode}, \mathsf{true}) \land \mathsf{arem}(\mathtt{INC}) \}$
 5    p.next := mynode;
      $\{ (P_0(\mathtt{p}, \mathtt{mynode}, \mathtt{mynode}, \mathsf{true}) \lor P_1(\mathtt{mynode}, \mathsf{false})) \land \mathsf{arem}(\mathtt{INC}) \}$
 6    b := mynode.locked;
      $\{ (P_0(\mathtt{p}, \mathtt{mynode}, \mathtt{mynode}, \mathsf{true}) \land \mathtt{b} \lor P_1(\mathtt{mynode}, \mathsf{false})) \land \mathsf{arem}(\mathtt{INC}) \}$
      $\{ (\mathtt{p} = p) \land (\mathtt{mynode} = x) \land (P_0(p, x, x, \mathsf{true}) \land \mathtt{b} \lor P_1(x, \mathsf{false})) \land \mathsf{arem}(\mathtt{INC}) \land \Diamond(\mathsf{bet}(\mathtt{b})) \}$
 7    while (b) {
 8      b := mynode.locked;
 9    }
10  }
    $\{ P_1(\mathtt{mynode}, \mathsf{false}) \land \mathsf{arem}(\mathtt{INC}) \}$
11   <acquired := true>;
    $\{ \mathsf{resource} * P_2(\mathtt{mynode}, \mathsf{false}) \land \mathsf{arem}(\mathtt{INC}) \}$
12  ...
```

Here the loop at lines 7-9 is verified in Figure 42.

Figure 41. Proof outline.

$$p' \overset{\text{def}}{=} (P_1(x, \mathsf{false}) \wedge \Diamond(0) \vee P_0(p, x, x, \mathsf{true}) \wedge \Diamond(\mathsf{bet}(\mathtt{b}))) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC})$$

$$\mathcal{D}'_{\mathsf{t}} \overset{\text{def}}{=} (\forall \mathsf{t}', ta, tl.\ dp1_{\mathsf{t}}(\mathsf{t}', ta, tl :: t_0) \rightsquigarrow dq1_{\mathsf{t}}(\mathsf{t}', ta, tl :: t_0)) \wedge (\forall tl_1, ta, tl_2.\ dp2_{\mathsf{t}}(tl_1, ta, tl_2 :: t_0) \rightsquigarrow dq2_{\mathsf{t}}(tl_1, ta, tl_2 :: t_0))$$

$$J_{\mathsf{t}} \overset{\text{def}}{=} (P_0(p, x, x, \mathsf{true}) \vee P_1(x, \mathsf{false})) \wedge (t_0 = \mathsf{t})\ , \text{ where } t_0 \text{ is a logical variable}$$

$$Q_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}})$$

$$G' \overset{\text{def}}{=} [I]$$

$$\begin{aligned} &f(\mathfrak{S}) = k \text{ iff} \\ &\quad \exists tl_1, tl_2, S_1, S_2.\ (\mathfrak{S} \models P_0(tl_1, tl_2, p, x, x, \mathsf{true}, S_1, S_2)) \wedge (k = \mathsf{len}(tl_1) + |S_1|) \\ &\quad \vee (\mathfrak{S} \models P_1(x, \mathsf{false})) \wedge (k = 0) \end{aligned}$$

$$\mathsf{len}(\epsilon) \overset{\text{def}}{=} 0 \qquad \mathsf{len}(\mathsf{t} :: tl) \overset{\text{def}}{=} 1 + \mathsf{len}(tl) \qquad |\emptyset| \overset{\text{def}}{=} 0 \qquad |S \uplus \{x\}| \overset{\text{def}}{=} |S| + 1$$

$$\begin{aligned} &(\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge (P_0(p, x, x, \mathsf{true}) \vee P_1(x, \mathsf{false})) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bet}(\mathtt{b})) \wedge Q * \mathsf{true} \\ &\Longrightarrow ((\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge P_1(x, \mathsf{false}) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(0)) * (\Diamond(1) \wedge \mathsf{emp}) \end{aligned}$$

```
      { $p'$ }
      { $(P_1(x, \mathsf{false}) \wedge \Diamond(0) \vee P_0(p, x, x, \mathsf{true}) \wedge \Diamond(\mathsf{bet}(\mathtt{b}))) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathtt{b} \wedge \mathsf{arem}(\mathtt{INC})$ }
8       b := mynode.locked;
      { $(P_0(p, x, x, \mathsf{true}) \wedge \mathtt{b} \vee P_1(x, \mathsf{false})) \wedge (\mathtt{p} = p) \wedge (\mathtt{mynode} = x) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{bet}(\mathtt{b}))$ }
```

We can prove:

(1) $J \Rightarrow I$; $Q \Rightarrow I$; $\mathsf{Sta}(J, G' \vee R)$.

(2) $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ (where picking $\mathcal{D}'$ needed in the WHL rule as above).

Figure 42. Proof outline – the loop at lines 7-9.

```
11  ...
   { $\mathsf{resource} * P_2(\mathtt{mynode}, \mathsf{false}) \wedge \mathsf{arem}(\mathtt{INC})$ }
12  r := x; x := r + 1;
   { $(\mathtt{x} = \mathtt{X} + 1) * P_2(\mathtt{mynode}, \mathsf{false}) \wedge \mathsf{arem}(\mathtt{INC})$ }
13  s := mynode.next;
   { $\exists tl, y.\ (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, y) \wedge ((\mathtt{s} = \mathtt{null}) \vee (\mathtt{s} = y)) \wedge \mathsf{arem}(\mathtt{INC})$ }
14  if (s = null) {
     { $(\mathtt{x} = \mathtt{X} + 1) * P_2(\mathtt{mynode}, \mathsf{false}) \wedge (\mathtt{s} = \mathtt{null}) \wedge \mathsf{arem}(\mathtt{INC})$ }
15    <nos := cas(&tail, mynode, null);  if(nos) { tq := getTail(tq); acquired := false; } >;
     { $(\mathtt{nos} \wedge P \wedge (\mathtt{s} = \mathtt{null}) \wedge \mathsf{arem}(\mathbf{skip})) \vee ((\neg\mathtt{nos}) \wedge \exists tl, y.\ (tl \neq \epsilon) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, y) \wedge \mathsf{arem}(\mathtt{INC}))$ }
16    if (!nos) {
       { $\exists tl, y.\ (tl \neq \epsilon) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, y) \wedge \mathsf{arem}(\mathtt{INC})$ }
17      s := mynode.next;
       { $\exists tl, y.\ (tl \neq \epsilon) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, y) \wedge ((\mathtt{s} = \mathtt{null}) \vee (\mathtt{s} = y)) \wedge \mathsf{arem}(\mathtt{INC})$ }
       { $\exists tl, y.\ (tl \neq \epsilon) \wedge (\mathtt{mynode} = x) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge ((\mathtt{s} = \mathtt{null}) \vee (\mathtt{s} = y)) \wedge \mathsf{arem}(\mathtt{INC}) \wedge \Diamond(\mathsf{pez}(\mathtt{s}))$ }
18      while (s = null) {
19        s := mynode.next;
20      }
21    }
22  }
   { $(P \wedge (\mathtt{s} = \mathtt{null}) \wedge \mathsf{arem}(\mathbf{skip})) \vee (\exists tl.\ (tl \neq \epsilon) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, \mathtt{s}) \wedge (\mathtt{s} \neq \mathtt{null}) \wedge \mathsf{arem}(\mathtt{INC}))$ }
23  if (s != null) {
     { $\exists tl.\ (tl \neq \epsilon) \wedge (\mathtt{x} = \mathtt{X} + 1) * P_2(tl, \mathtt{mynode}, \mathsf{false}, \mathtt{s}) \wedge (\mathtt{s} \neq \mathtt{null}) \wedge \mathsf{arem}(\mathtt{INC})$ }
24    <s.locked := false;  tq := getTail(tq); acquired := false>;
     { $P(\mathtt{mynode}, \_) \wedge \mathsf{arem}(\mathbf{skip})$ }
25    mynode.next := null;
26  }
   { $P \wedge \mathsf{arem}(\mathbf{skip})$ }
```

Here the loop at lines 18-20 is verified in Figure 44.

Figure 43. Proof outline (continued).

$$p'' \overset{\text{def}}{=} \exists tl, y.\ (tl \neq \epsilon) \wedge (\texttt{mynode} = x) \wedge (\texttt{x} = \texttt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge (\texttt{s} = \texttt{null}) \wedge \mathsf{arem}(\texttt{INC})$$
$$\wedge\, (y \neq \texttt{null} \wedge \Diamond(0) \vee y = \texttt{null} \wedge \Diamond(\mathsf{pez}(\texttt{s})))$$

$$\mathcal{D}''_{\mathsf{t}} \overset{\text{def}}{=} (\forall tl_1, ta, tl_2.\ dp2_{\mathsf{t}}(tl_1 :: t_0, ta, tl_2) \leadsto dq2_{\mathsf{t}}(tl_1 :: t_0, ta, tl_2))$$

$J'_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, y.\ (tl \neq \epsilon) \wedge P_2(tl, x, \mathsf{false}, y) \wedge (t_0 = \mathsf{t})$, where t_0 is a logical variable

$$Q'_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}})$$

$$G' \overset{\text{def}}{=} [I]$$

$$f'(\mathfrak{S}) = k \text{ iff } \exists tl, y.\ (tl \neq \epsilon) \wedge (\mathfrak{S} \models P_2(tl, x, \mathsf{false}, y)) \wedge (k = \mathsf{pez}(y))$$

$$(\exists tl, y.\ (tl \neq \epsilon) \wedge (\texttt{mynode} = x) \wedge (\texttt{x} = \texttt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge (\texttt{s} = \texttt{null}) \wedge \mathsf{arem}(\texttt{INC}) \wedge \Diamond(\mathsf{pez}(\texttt{s})) \wedge Q' * \mathsf{true})$$
$$\Longrightarrow (\exists tl, y.\ (tl \neq \epsilon) \wedge (\texttt{mynode} = x) \wedge (\texttt{x} = \texttt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge (y \neq \texttt{null}) \wedge (\texttt{s} = \texttt{null}) \wedge \mathsf{arem}(\texttt{INC}) \wedge \Diamond(0))$$
$$* (\Diamond(1) \wedge \mathsf{emp})$$

$\{ p'' \}$
$\left\{ \begin{array}{l} \exists tl, y.\ (tl \neq \epsilon) \wedge (\texttt{mynode} = x) \wedge (\texttt{x} = \texttt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge (\texttt{s} = \texttt{null}) \wedge \mathsf{arem}(\texttt{INC}) \\ \quad \wedge (y \neq \texttt{null} \wedge \Diamond(0) \vee y = \texttt{null} \wedge \Diamond(\mathsf{pez}(\texttt{s}))) \end{array} \right\}$

```
19  s := mynode.next;
```

$\{ \exists tl, y.\ (tl \neq \epsilon) \wedge (\texttt{mynode} = x) \wedge (\texttt{x} = \texttt{X} + 1) * P_2(tl, x, \mathsf{false}, y) \wedge ((\texttt{s} = \texttt{null}) \vee (\texttt{s} = y)) \wedge \mathsf{arem}(\texttt{INC}) \wedge \Diamond(\mathsf{pez}(\texttt{s})) \}$

We can prove:

(1) $J' \Rightarrow I; Q' \Rightarrow I; \mathsf{Sta}(J', G' \vee R)$.

(2) $J' \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f'} Q')$ (where picking $\mathcal{D}'$ needed in the WHL rule as above $\mathcal{D}''$).

Figure 44. Proof outline (continued) – the loop at lines 18-20.

Abstract operations:

```
ENQ(V) {  < Q := Q :: V; > }

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

```
struct Node {
  int data;
  struct Node *next;
}
struct Lock {
  int lowner, lnext;
  int[] ticket; //initially all -1
}
struct Queue {
  struct Node *Head;
  struct Node *Tail;
  struct Lock *Hlock, Tlock;
}

initialize(){
  Head := cons(0, null);
  Tail := Head;
  Hlock.lowner := Hlock.lnext := Tlock.lowner := Tlock.lnext := 0;
}
```

```
enq(v) {
  local x, i, o;
  x := cons(v, null);
  <i := getAndInc(Tlock.lnext); Tlock.ticket_i := cid>;
  <o := Tlock.lowner>;
  while (i <> o) {
    <o := Tlock.lowner>;
  }
  Tail.next := x;
  Tail := x;
  <Tlock.lowner++>;
}
```

```
int deq() {
  local h, s, v, i, o;
  <i := getAndInc(Hlock.lnext); Hlock.ticket_i := cid>;
  <o := Hlock.lowner>;
  while (i <> o) {
    <o := Hlock.lowner>;
  }
  h := Head;
  s := h.next;
  if (s = null) {
    <Hlock.lowner++>;
    v := EMPTY;
  } else {
    v := s.data;
    Head := s;
    <Hlock.lowner++>;
    dispose(h);
  }
  return v;
}
```

Figure 45. Two-lock queue implementation.

C.5 Two-lock queue with ticket lock

In Figure 45, we show the abstract code of the queue algorithm and the concrete implementation of the two-lock queue with auxiliary code (in red). In the concrete implementation, the two locks are implemented using ticket lock. In the algorithm, the queue is implemented as a singly-linked list with the `Head` and `Tail` pointers and a sentinel node pointed to by `Head`. The `enq` method inserts a new node at the tail of the queue. The `deq` method replaces the sentinel node by its next node and returns the value in the new sentinel. The `Tail` and `Head` pointers are protected by two locks `Tlock` and `Hlock` respectively. At any time at most one `enq` thread and one `deq` thread can access the queue. But an `enq` thread and a `deq` thread do not need to wait for each other.

Since the locks are implemented using the ticket lock algorithm, we introduce the auxiliary `tickets` to verify the code (as in the proof for the counter with ticket lock). Note here we do not need the auxiliary `wait` bits as in the proof for the counter with ticket lock, because now we do not have ownership transfers of resource when the lock is acquired/released.

Fig. 46 defines the precise invariant and the precondition. Fig. 47 defines the rely and guarantee conditions and the definite action. The definite action for the whole object is defined as the conjunction of the definite actions on the two locks: $\mathcal{D} \overset{\text{def}}{=} dT \wedge dH$, where dT (or dH) says that the thread must eventually release the lock if it has acquired `Tlock` (or `Hlock`).

Fig. 48 and Fig. 50 show the proof outlines. The proof of the algorithm in our logic combines its linearizability proof [21] and the proofs on the progress properties of the ticket lock implementation, which are similar to the above proofs for the counter object with ticket lock. It also relies on the fact that if a thread is holding a lock, it will not request the other lock. Note that the proofs for the loops of acquiring the locks follow the proofs for the counter with ticket lock (see Sec. C.1).

$tl ::= \epsilon \mid \mathsf{t} :: tl$

$\mathsf{list2set}(\epsilon) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{list2set}(\mathsf{t} :: tl) \stackrel{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(tl)$

$I \stackrel{\text{def}}{=} \exists h, z, tl_T.\ (\mathtt{Head} = h) * (\mathtt{Tail} = z) * \mathsf{queue}(tl_T, h, z) * \mathsf{lock}(\mathtt{Hlock}) * \mathsf{lock}(tl_T, \mathtt{Tlock})$

$\mathsf{queue}(tl_T, h, z) \stackrel{\text{def}}{=}$
$\quad \exists v_d, A.\ (\mathtt{Q} = A) * (\mathsf{unlag}(h, z, v_d :: A) \vee (\mathsf{lag}(h, z, \_, v_d :: A) \wedge (tl_T \neq \epsilon)) \vee (\mathsf{cross}(h, v_d :: A) \wedge (tl_T \neq \epsilon)))$

$\mathsf{unlag}(h, z, A) \stackrel{\text{def}}{=} \exists v, A'.\ (A = A' :: v) \wedge \mathsf{ls}(h, A', z) * \mathsf{N}(z, v, \mathtt{null})$

$\mathsf{lag}(h, z, x, A) \stackrel{\text{def}}{=} \exists v, v', A'.\ (A = A' :: v :: v') \wedge \mathsf{ls}(h, A', z) * \mathsf{N2}(z, v, x, v', \mathtt{null})$

$\mathsf{cross}(h, A) \stackrel{\text{def}}{=} \exists v.\ (A = v :: \epsilon) \wedge \mathsf{N}(h, v, \mathtt{null})$

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

$\mathsf{N}(p, v, y) \stackrel{\text{def}}{=} (p.\mathtt{data} = v) * (p.\mathtt{next} = y) \qquad \mathsf{N2}(p, v, y, v', z) \stackrel{\text{def}}{=} \mathsf{N}(p, v, y) * \mathsf{N}(y, v', z)$

$\mathsf{lock}(l) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock}(tl, l) \qquad \mathsf{lock}(tl, l) \stackrel{\text{def}}{=} \exists n_1, n_2.\ \mathsf{lock}(tl, l, n_1, n_2)$

$\mathsf{lock}(tl, l, n_1, n_2) \stackrel{\text{def}}{=}$
$\quad ((l.\mathtt{lowner} = n_1) * (l.\mathtt{lnext} = n_2) \wedge (n_1 \leq n_2)) * \mathsf{tickets}(l, 0, n_1) * \mathsf{tickets}(tl, l, n_1, n_2) * \mathsf{tickets\_new}(l, n_2)$

$\mathsf{tickets}(l, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{tickets}(tl, l, n_1, n_2)$

$\mathsf{tickets}(tl, l, n_1, n_2) \stackrel{\text{def}}{=}$
$\quad (tl = \epsilon) \wedge (n_1 = n_2) \wedge \mathsf{emp} \ \vee\ \exists \mathsf{t}, tl'.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin \mathsf{list2set}(tl')) \wedge (l.\mathtt{ticket}_{n_1} = \mathsf{t}) * \mathsf{tickets}(tl', l, n_1 + 1, n_2)$

$\mathsf{tickets\_new}(l, n_2) \stackrel{\text{def}}{=} (\circledast_{i \geq n_2} l.\mathtt{ticket}_i = -1)$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists h, z, tl_T.\ (\mathtt{Head} = h) * (\mathtt{Tail} = z) * \mathsf{queue}(tl_T, h, z) * \mathsf{lock\_irr}_{\mathsf{t}}(\mathtt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \mathtt{Tlock})$

$\mathsf{lock\_irr}_{\mathsf{t}}(l) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, l) \qquad \mathsf{lock\_irr}_{\mathsf{t}}(tl, l) \stackrel{\text{def}}{=} \exists n_1, n_2.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, l, n_1, n_2)$

$\mathsf{lock\_irr}_{\mathsf{t}}(tl, l, n_1, n_2) \stackrel{\text{def}}{=} \mathsf{lock}(tl, l, n_1, n_2) \wedge (\mathsf{t} \notin \mathsf{list2set}(tl))$

Figure 46. Invariant and precondition of the two-lock queue.

$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}'\neq\mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \overset{\text{def}}{=} (\mathit{Enq}_{\mathsf{t}} \vee \mathit{Swing}_{\mathsf{t}} \vee \mathit{Deq}_{\mathsf{t}} \vee \mathit{ReqLockH}_{\mathsf{t}} \vee \mathit{UnlockH}_{\mathsf{t}} \vee \mathit{ReqLockT}_{\mathsf{t}} \vee \mathit{UnlockT}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Enq}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, x, y, A, v, v'.\ [\mathsf{lock}(\mathsf{t}::tl, \mathtt{Tlock}) * (\mathtt{Tail} = x)]$
$\quad * ((\mathsf{N}(x, v, \mathtt{null}) * (\mathtt{Q} = A)) \ltimes (\mathsf{N2}(x, v, y, v', \mathtt{null}) * (\mathtt{Q} = A::v'))$

$\mathit{Swing}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, x, v.\ [\mathsf{lock}(\mathsf{t}::tl, \mathtt{Tlock}) * \mathsf{N}(x, v, \mathtt{null})] * ((\mathtt{Tail} = \_) \ltimes (\mathtt{Tail} = x))$

$\mathit{Deq}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, x, y, z, v, v', A.\ [\mathsf{lock}(\mathsf{t}::tl, \mathtt{Hlock})]$
$\quad * (((\mathtt{Head} = x) * \mathsf{N2}(x, v, y, v', z) * (\mathtt{Q} = v'::A)) \ltimes ((\mathtt{Head} = y) * \mathsf{N}(y, v', z) * (\mathtt{Q} = A)))$

$\mathit{ReqLockH}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\ [\mathsf{lock\_irr}_{\mathsf{t}}(\mathtt{Tlock})] * (\mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Hlock}, n_1, n_2) \ltimes \mathsf{lock}(tl::\mathsf{t}, \mathtt{Hlock}, n_1, n_2+1))$

$\mathit{UnlockH}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\ [\mathsf{lock\_irr}_{\mathsf{t}}(\mathtt{Tlock})] * (\mathsf{lock}(\mathsf{t}::tl, \mathtt{Hlock}, n_1, n_2) \ltimes \mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Hlock}, n_1+1, n_2))$

$\mathit{ReqLockT}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists tl, n_1, n_2.\ [\mathsf{lock\_irr}_{\mathsf{t}}(\mathtt{Hlock})] * (\mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Tlock}, n_1, n_2) \ltimes \mathsf{lock}(tl::\mathsf{t}, \mathtt{Tlock}, n_1, n_2+1))$

$\mathit{UnlockT}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists z, v, tl, n_1, n_2.\ [\mathsf{lock\_irr}_{\mathsf{t}}(\mathtt{Hlock}) * (\mathtt{Tail} = z) * \mathsf{N}(z, v, \mathtt{null})]$
$\quad * (\mathsf{lock}(\mathsf{t}::tl, \mathtt{Tlock}, n_1, n_2) \ltimes \mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Tlock}, n_1+1, n_2))$

$\mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} (\forall n_1.\ dpH_{\mathsf{t}}(n_1) \leadsto dqH_{\mathsf{t}}(n_1)) \wedge (\forall n_1.\ dpT_{\mathsf{t}}(n_1) \leadsto dqT_{\mathsf{t}}(n_1))$

$dpH_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2.\ \mathsf{lock}(\mathsf{t}::tl, \mathtt{Hlock}, n_1, n_2) * \mathsf{true} \wedge I$

$dqH_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Hlock}, n_1+1, n_2) * \mathsf{true} \wedge I$

$dpT_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2.\ \mathsf{lock}(\mathsf{t}::tl, \mathtt{Tlock}, n_1, n_2) * \mathsf{true} \wedge I$

$dqT_{\mathsf{t}}(n_1) \overset{\text{def}}{=} \exists tl, n_2.\ \mathsf{lock\_irr}_{\mathsf{t}}(tl, \mathtt{Tlock}, n_1+1, n_2) * \mathsf{true} \wedge I$

Figure 47. Rely, guarantee and definite actions of the two-lock queue.

$\mathsf{tlocked}_{tl_1,\mathsf{t},tl_2}(l, n_1, n, n_2) \overset{\text{def}}{=}$
$\quad(\mathsf{list2set}(tl_1) \cap \mathsf{list2set}(\mathsf{t}::tl_2) = \emptyset) \wedge ((l.\texttt{lowner} = n_1) * (l.\texttt{lnext} = n_2) \wedge (n_1 \le n < n_2))$
$\quad * \mathsf{tickets}(l, 0, n_1) * \mathsf{tickets}(tl_1, l, n_1, n) * \mathsf{tickets}(\mathsf{t}::tl_2, l, n, n_2) * \mathsf{tickets\_new}(l, n_2)$

$P_0(n_1, n, n_2) \overset{\text{def}}{=} \exists tl_1.\ P_0(tl_1, n_1, n, n_2)$ $\qquad P_3(n_1, n, n_2) \overset{\text{def}}{=} \exists \mathsf{t}', tl_1.\ P_0(\mathsf{t}'::tl_1, n_1, n, n_2)$

$P_0(tl_1, n_1, n, n_2) \overset{\text{def}}{=}$
$\quad\exists h, z, tl_2.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(tl_1, h, z) * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{tlocked}_{tl_1,\mathsf{t},tl_2}(\texttt{Tlock}, n_1, n, n_2)$

$P_1(n_1, n_2) \overset{\text{def}}{=}$
$\quad\exists h, z, v_d, A, tl.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * \mathsf{unlag}(h, z, v_d::A) * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock}(\mathsf{t}::tl, \texttt{Tlock}, n_1, n_2)$

$P_2(x, n_1, n_2) \overset{\text{def}}{=}$
$\quad\exists h, z, v_d, A, tl.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * (\mathsf{lag}(h, z, x, v_d::A) \vee (\mathsf{cross}(h, v_d::A) \wedge (h = x)))$
$\quad * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock}(\mathsf{t}::tl, \texttt{Tlock}, n_1, n_2)$

```
enq(v) {
 1  local x, i, o;
    { $P \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
 2  x := cons(v, null);
    { $P * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
 3  <i := getAndInc(Tlock.lnext);  Tlock.ticket_i := cid>;
    { $\exists n_1, n, n_2.\ P_0(n_1, n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
 4  <o := Tlock.lowner>;
    { $\exists n_1, n, n_2.\ P_0(n_1, n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge (\texttt{o} \le n_1) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
    { $\exists n_1, n_2.\ P_0(n_1, n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge (\texttt{o} \le n_1) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \Diamond(n - \texttt{o})$ }
 5  while (i <> o) {
 6    <o := Tlock.lowner>;
 7  }
    { $\exists n_1, n_2.\ P_1(n_1, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n_1) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
 8  Tail.next := x;
    { $\exists n_1, n_2.\ P_2(\texttt{x}, n_1, n_2) \wedge \mathsf{arem}(\textbf{skip})$ }
 9  Tail := x;
    { $\exists n_1, n_2.\ P_1(n_1, n_2) \wedge \mathsf{arem}(\textbf{skip})$ }
10  <Tlock.lowner++>;
    { $P \wedge \mathsf{arem}(\textbf{skip})$ }
}
```

Here the loop at lines 5-7 is verified in Figure 49.

Figure 48. Proof outline for enq.

$p' \overset{\text{def}}{=} \exists n_1, n_2.\ (P_1(n, n_2) \wedge \Diamond(n - (\texttt{o} + 1)) \vee P_3(n_1, n, n_2) \wedge \Diamond(n - \texttt{o})) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null})$
$\qquad \wedge (\texttt{o} \le n_1 \le n = \texttt{i}) \wedge (\texttt{o} \ne \texttt{i}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$

$\mathcal{D}'_{\mathsf{t}} \overset{\text{def}}{=} (\forall n_1.\ dpT_{\mathsf{t}}(n_1) \leadsto dqT_{\mathsf{t}}(n_1))$

$J \overset{\text{def}}{=} \exists n_1, n_2.\ P_0(n_1, n, n_2)$

$Q \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D})$

$G' \overset{\text{def}}{=} [I]$

$f(\mathfrak{S}) = k$ iff $\mathfrak{S} \models (n - \texttt{Tlock.lowner} = k)$

$(\exists n_1, n_2.\ P_0(n_1, n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge (\texttt{o} \le n_1) \wedge (\texttt{i} \ne \texttt{o}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \Diamond(n - \texttt{o}) \wedge Q * \mathsf{true})$
$\Longrightarrow (\exists n_2.\ P_1(n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge (\texttt{o} < n) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \Diamond(n - (\texttt{o} + 1))) * (\Diamond(1) \wedge \mathsf{emp})$

```
    { $p'$ }
    { $\exists n_1, n_2.\ (P_1(n, n_2) \wedge \Diamond(n - (\texttt{o} + 1)) \vee P_3(n_1, n, n_2) \wedge \Diamond(n - \texttt{o})) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null})$
      $\wedge (\texttt{o} \le n_1 \le n = \texttt{i}) \wedge (\texttt{o} \ne \texttt{i}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V})$ }
6     <o := Tlock.lowner>;
    { $\exists n_1, n_2.\ P_0(n_1, n, n_2) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge (\texttt{i} = n) \wedge (\texttt{o} \le n_1) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \Diamond(n - \texttt{o})$ }
```

We can prove:

(1) $J \Rightarrow I$; $Q \Rightarrow I$; $\mathsf{Sta}(J, G' \vee R)$.

(2) $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ (where picking $\mathcal{D}'$ needed in the WHL rule as above).

Figure 49. Proof outline for enq – the loop at lines 5-7.

$P'_0(n_1, n, n_2) \stackrel{\text{def}}{=} \exists tl_1.\, P'_0(tl_1, n_1, n, n_2)$ $\qquad$ $P'_5(n_1, n, n_2) \stackrel{\text{def}}{=} \exists \mathsf{t}', tl_1.\, P'_0(\mathsf{t}' :: tl_1, n_1, n, n_2)$

$P'_0(tl_1, n_1, n, n_2) \stackrel{\text{def}}{=}$
$\quad \exists h, z, tl_2, tl_T.\, (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(tl_T, h, z) * \mathsf{tlocked}_{tl_1, \mathsf{t}, tl_2}(\texttt{Hlock}, n_1, n, n_2) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \texttt{Tlock})$

$P'_1(h) \stackrel{\text{def}}{=} \exists n_1, n_2.\, P'_1(h, n_1, n_2)$ $\qquad$ $P'_3(h, s) \stackrel{\text{def}}{=} \exists v.\, P'_3(h, s, v)$

$P'_1(h, n_1, n_2) \stackrel{\text{def}}{=}$
$\quad \exists z, tl, tl_T.\, (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(tl_T, h, z) * \mathsf{lock}(\mathsf{t} :: tl, \texttt{Hlock}, n_1, n_2) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \texttt{Tlock})$

$P'_2(h, s) \stackrel{\text{def}}{=}$
$\quad \exists z, A, tl, tl_T.\, (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * \mathsf{N}(h, \_, s) * (\mathsf{unlag}(s, z, A) \vee \mathsf{lag}(s, z, \_, A) \wedge (tl_T \neq \epsilon))$
$\quad * \mathsf{lock}(\mathsf{t} :: tl, \texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \texttt{Tlock})$

$P'_3(h, s, v) \stackrel{\text{def}}{=}$
$\quad \exists tl, tl_T.\, (\texttt{Head} = h) * (\texttt{Tail} = h) * (\texttt{Q} = v :: \epsilon) * \mathsf{N2}(h, \_, s, v, \texttt{null}) * \mathsf{lock}(\mathsf{t} :: tl, \texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \texttt{Tlock}) \wedge (tl_T \neq \epsilon)$

$P'_4(h, s, v) \stackrel{\text{def}}{=}$
$\quad \exists x, z, v, A, tl, tl_T.\, (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = v :: A) * \mathsf{N2}(h, \_, s, v, x) * \mathsf{lock}(\mathsf{t} :: tl, \texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t}}(tl_T, \texttt{Tlock})$
$\quad * ((s = z) \wedge (x = \texttt{null}) \wedge (A = \epsilon) \vee \mathsf{unlag}(x, z, A)$
$\quad\quad \vee (s = z) \wedge \mathsf{N}(x, v', \texttt{null}) \wedge (A = v' :: \epsilon) \wedge (tl_T \neq \epsilon) \vee \mathsf{lag}(x, z, \_, A) \wedge (tl_T \neq \epsilon))$

```
int deq() {
 1  local h, s, v, i, o;
    { $P \wedge \mathsf{arem}(\texttt{DEQ})$ }
 2  <i := getAndInc(Hlock.lnext); Hlock.ticket_i := cid>;
    { $\exists n_1, n, n_2.\, P'_0(n_1, n, n_2) \wedge (\texttt{i} = n) \wedge \mathsf{arem}(\texttt{DEQ})$ }
 3  <o := Hlock.lowner>;
    { $\exists n_1, n, n_2.\, P'_0(n_1, n, n_2) \wedge (\texttt{i} = n) \wedge (\texttt{o} \leq n_1) \wedge \mathsf{arem}(\texttt{DEQ})$ }
    { $\exists n_1, n_2.\, P'_0(n_1, n, n_2) \wedge (\texttt{i} = n) \wedge (\texttt{o} \leq n_1) \wedge \mathsf{arem}(\texttt{DEQ}) \wedge \Diamond(n - \texttt{o})$ }
 4  while (i <> o) {
 5    <o := Hlock.lowner>;
 6  }
    { $\exists h.\, P'_1(h) \wedge \mathsf{arem}(\texttt{DEQ})$ }
 7  h := Head;
    { $P'_1(\texttt{h}) \wedge \mathsf{arem}(\texttt{DEQ})$ }
 8  s := h.next;
    { $((\texttt{s} = \texttt{null}) \wedge P'_1(\texttt{h}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY})) \vee (P'_2(\texttt{h}, \texttt{s}) \vee P'_3(\texttt{h}, \texttt{s})) \wedge \mathsf{arem}(\texttt{DEQ})$ }
 9  if (s = null) {
      { $P'_1(\texttt{h}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY})$ }
10    <Hlock.lowner++>;
      { $P \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY})$ }
11    v := EMPTY;
12  } else {
      { $(P'_2(\texttt{h}, \texttt{s}) \vee P'_3(\texttt{h}, \texttt{s})) \wedge \mathsf{arem}(\texttt{DEQ})$ }
13    v := s.data;
      { $(P'_4(\texttt{h}, \texttt{s}, \texttt{v}) \vee P'_3(\texttt{h}, \texttt{s}, \texttt{v})) \wedge \mathsf{arem}(\texttt{DEQ})$ }
14    Head := s;
      { $P'_1(\texttt{s}) * \mathsf{N}(\texttt{h}, \_, \texttt{s}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V})$ }
15    <Hlock.lowner++>;
      { $P * \mathsf{N}(\texttt{h}, \_, \texttt{s}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V})$ }
16    dispose(h);
17  }
    { $P \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V})$ }
18  return v;
}
```

Here the loop at lines 4-6 is verified in Figure 51.

Figure 50. Proof outline for deq.

$$p'' \stackrel{\text{def}}{=} \exists h, n_1, n_2.\, (P'_1(h, n, n_2) \wedge \Diamond(n - (\texttt{o} + 1)) \vee P'_5(n_1, n, n_2) \wedge \Diamond(n - \texttt{o})) \\ \wedge (\texttt{o} \leq n_1 \leq n = \texttt{i}) \wedge (\texttt{i} \neq \texttt{o}) \wedge \mathsf{arem}(\texttt{DEQ})$$

$$\mathcal{D}''_{\mathtt{t}} \stackrel{\text{def}}{=} (\forall n_1.\, dpH_{\mathtt{t}}(n_1) \rightsquigarrow dqH_{\mathtt{t}}(n_1))$$

$$J' \stackrel{\text{def}}{=} \exists n_1, n_2.\, P'_0(n_1, n, n_2)$$

$$Q' \stackrel{\text{def}}{=} \mathsf{Enabled}(\mathcal{D})$$

$$G' \stackrel{\text{def}}{=} [I]$$

$$f'(\mathfrak{S}) = k \text{ iff } \mathfrak{S} \models (n - \texttt{Hlock.lowner} = k)$$

$$(\exists n_1, n_2.\, P'_0(n_1, n, n_2) \wedge (\texttt{i} = n) \wedge (\texttt{o} \leq n_1) \wedge (\texttt{i} \neq \texttt{o}) \wedge \mathsf{arem}(\texttt{DEQ}) \wedge \Diamond(n - \texttt{o}) \wedge Q' * \mathsf{true}) \\ \Longrightarrow (\exists h, n_2.\, P'_1(h, n, n_2) \wedge (\texttt{i} = n) \wedge (\texttt{o} < n) \wedge \mathsf{arem}(\texttt{DEQ}) \wedge \Diamond(n - (\texttt{o} + 1))) * (\Diamond(1) \wedge \mathsf{emp})$$

```
  {$p''$}
  {$\exists h, n_1, n_2.\, (P'_1(h, n, n_2) \wedge \Diamond(n - (\texttt{o} + 1)) \vee P'_5(n_1, n, n_2) \wedge \Diamond(n - \texttt{o}))$
   $\wedge (\texttt{o} \leq n_1 \leq n = \texttt{i}) \wedge (\texttt{i} \neq \texttt{o}) \wedge \mathsf{arem}(\texttt{DEQ})$}
5  <o := Hlock.lowner>;
  {$\exists n_1, n_2.\, P'_0(n_1, n, n_2) \wedge (\texttt{i} = n) \wedge (\texttt{o} \leq n_1) \wedge \mathsf{arem}(\texttt{DEQ}) \wedge \Diamond(n - \texttt{o})$}
```

We can prove:

(1) $J' \Rightarrow I$; $Q' \Rightarrow I$; $\mathsf{Sta}(J', G' \vee R)$.

(2) $J' \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f'} Q')$ (where picking $\mathcal{D}'$ needed in the WHL rule as above $\mathcal{D}''$).

Figure 51. Proof outline for `deq` – the loop at lines `4-6`.

C.6 Lock-coupling list with ticket lock

In Figure 52, we show the abstract code of the set algorithm and the concrete implementation of the lock-coupling list with auxiliary code (in red and purple). In the concrete implementation, the node locks are all implemented using ticket lock. The algorithm implements an abstract set with `add` and `rmv` operations. The concrete list is an ordered singly-linked list with the `Head` pointer and two sentinel nodes at the two ends of the list containing the smallest and the biggest values respectively. Each list node is associated with a lock. Traversing the list uses hand-over-hand locking: the lock on one node is not released until its successor is locked. `add(e)` inserts a new node with value `e` in the appropriate position while holding the lock of its predecessor. `rmv(e)` redirects the predecessor's pointer when both the node to be removed and its predecessor are locked.

To verify the code, we need some auxiliary structures. Since the locks are implemented using the ticket lock algorithm, we introduce the auxiliary `tickets` for the lock of each node (as in the previous proof for the two-lock queue with ticket locks). Also, we introduce the auxiliary `toadd` bit for each thread to indicate whether the thread attempts to do the `add` operation. $\mathtt{toadd}_\mathtt{t}$ is `true` if the thread `t` is calling `add` but has not really inserted the new node into the list. It is `false` if the thread `t` is not calling `add` or it has already inserted the new node. The auxiliary `tickets` and `toadd` bits are all shared. In addition, we introduce an auxiliary local variable `aux` to encode the metric for the loop of list traversal. Fig. 53 shows the auxiliary code, `metric(p)`, to compute the metric for the loop of list traversal. Here `p` points to the "predecessor" node during the list traversal Informally, `metric(p)` is the sum of the current maximal length of the remaining nodes to traverse (i.e., the number of the nodes from `p` to the end of the list) and the number of the environment threads accessing the remaining nodes which attempt to do `add` (i.e., these threads is going to add nodes, increasing the maximal length of the nodes remained for the current thread to traverse).

Fig. 54 defines the precise invariant and the precondition. The shared state contains the `toadd` bit for each thread and the list. Each list node contains a ticket lock. The predicate ss requires the concrete list should be sorted and its elements should constitute the abstract set.

Fig. 55 defines the rely and guarantee conditions and the definite action. In addition to the actions *add* and *rmv* which successfully inserts and removes a node, the thread may do the *ToAdd* action to set its `toadd` bit to `true` when it just starts its `add` operation. If the `add` fails (i.e., for the case when the value it attempts to insert has already in the list), the thread does the *FailedAdd* action that simply resets its `toadd` bit to `false`. The *ReqLockH* action requests the lock of the head node in the list. At that time, the thread must just starts its operation, thus it has not requested the lock of a list node. The *ReqLock2* action requests the lock of a node when the thread has already acquired the lock of the predecessor node. The *Unlock* action releases the lock of the predecessor node.

The difficulty of verifying the lock-coupling list algorithm lies in defining the definite action $\mathcal{D}$. Unlike the counter objects and the two-lock queues, in lock-coupling lists, the thread `t` holding a lock does not guarantee to release the lock because `t` may first try to acquire the lock of the successor node. In fact, the progress of `t` relies on the progress of the environment threads which start their list traversals earlier than the thread `t`. The nodes being visited by these environment threads might be visited by `t` in the future. Therefore we define the definite action $\mathcal{D}$ to describe the progress of the thread which is the first to start the list traversal, i.e., the thread accessing the node closest to the tail of the list. Informally, $\mathcal{D}_\mathtt{t}$ says that `t` eventually releases the lock of the node x, if `t` holds the lock and all the nodes remained to visit (the nodes from the successor of x to the end of the list) are either unlocked or locked by `t` itself.

The whole proof follows the basic linearizability proof of lock-coupling list (e.g., see [21]). Now we also verify the starvation-freedom of the object.

We first define the general well-founded order and the metrics for the loops in Fig. 56. We show the proof for the `add` operation in Fig. 57. and the proof for the `rmv` operation in Fig. 61. Note we define the Q needed for verifying the loops similarly as the case when $\mathcal{D}$ is enabled. Actually the outer loop of list traversal always terminates no matter whether $\mathcal{D}$ is enabled or not, because the number of the nodes remained to traverse (including the nodes possibly added in the future by the environment threads) must be decreasing, i.e., the metric function `metric(p)` defined in Fig. 53 must be decreasing at each loop iteration. For the inner loop of requesting the lock of a node `c`, if $\mathcal{D}$ is enabled, we know there is no thread accessing the node `c`, then the current thread must succeeds in acquiring the lock and the loop could terminate.

Below we explain how we define f needed for verifying the loops. As shown in Fig. 56, for any state during the execution of a loop, the metric contains two parts: *toadd*, the `toadd` bits of all the threads, and *L*, the list of the environment threads ahead of the current thread (including the threads which are ahead of the current thread and are requesting the lock of the same node). The well-founded order defined at the top of Fig. 56 says, the metric decreases if (1) the number of the environment threads in *L* decreases; or (2) the number of the environment threads which attempt to do `add` and are ahead of the current thread decreases; or (3) some environment thread ahead of the current thread gets closer to the end of the list. Thus the metric must decrease when a relevant environment thread progresses, and the key requirements $J \Rightarrow (R, G\colon \mathcal{D} \xrightarrow{f} Q)$ for verifying loops hold.

```
ADD(E) {                                RMV(E) {
  local R;                                local R;
  < if (E ∈ S) {                          < if (E ∉ S) {
      R := false;                             R := false;
    } else {                                } else {
      S := S ∪ {E};  R := true;               S := S \ {E};  R := true;
    }                                       }
  >                                       >
  return R;                               return R;
}                                       }
```

(a) Abstract operations.

```
bool[] toadd; //initially all false
                                        struct List {
struct Node {                             struct Node *Head;
  int lowner;                           }
  int lnext;
  int[] ticket; //initially all -1      initialize(){
  int data;                               Head := cons(0, 0, MIN, null);
  struct Node *next;                      Head.next := cons(0, 0, MAX, null);
}                                       }
```

```
add(e) {
  local p, c, x, pi, po, ci, co, u, r, aux;
  <p := Head; toadd_cid := true>;
  <pi := getAndInc(p.lnext); p.ticket_pi := cid>;
  <po := p.lowner>;
  while (pi <> po) {
    <po := p.lowner>;
  }
  c := p.next;
  <u := c.data; aux := metric(p)>;
  while (u < e) {
    <ci := getAndInc(c.lnext); c.ticket_ci := cid>;
    <co := c.lowner>;
    while (ci <> co) {
      <co := c.lowner>;
    }
    <p.lowner++>;
    p := c;
    c := p.next;
    <u := c.data; aux := metric(p)>;
  }
  if (u != e) {
    x := cons(0, 0, e, c);
    <p.next := x; toadd_cid := false>;
    r := true;
  } else {
    <r := false; toadd_cid := false>;
  }
  <p.lowner++>;
  return r;
}
```

```
rmv(e) {
  local p, c, n, pi, po, ci, co, u, r, aux;
  p := Head;
  <pi := getAndInc(p.lnext); p.ticket_pi := cid>;
  <po := p.lowner>;
  while (pi <> po) {
    <po := p.lowner>;
  }
  c := p.next;
  <u := c.data; aux := metric(p)>;
  while (u < e) {
    <ci := getAndInc(c.lnext); c.ticket_ci := cid>;
    <co := c.lowner>;
    while (ci <> co) {
      <co := c.lowner>;
    }
    <p.lowner++>;
    p := c;
    c := p.next;
    <u := c.data; aux := metric(p)>;
  }
  if (u = e) {
    <ci := getAndInc(c.lnext); c.ticket_ci := cid>;
    <co := c.lowner>;
    while (ci <> co) {
      <co := c.lowner>;
    }
    n := c.next;
    p.next := n;
    <p.lowner++>;
    dispose(c);
    r := true;
  } else {
    <p.lowner++>;
    r := false;
  }
  return r;
}
```

(b) Concrete implementations.

Figure 52. Lock-coupling list.

```
int metric(p){
  local n, l, s, o, t;
  n := p.next; l := 0; s := $\emptyset$;
  while (n <> null) {
    l++;
    o := n.lowner;
    while (o < n.lnext) {
      t := n.ticket$_{\mathtt{o}}$;
      if (toadd$_{\mathtt{t}}$ = true) s := s $\cup$ {t};
    }
    n := n.next;
  }
  l := l + |s|;
  return l;
}
```

$$
\begin{aligned}
|\emptyset| &\stackrel{\text{def}}{=} 0 \\
|S \cup \{x\}| &\stackrel{\text{def}}{=} |S| + 1 \\[1em]
\mathsf{len}(\epsilon) &\stackrel{\text{def}}{=} 0 \\
\mathsf{len}(v :: A) &\stackrel{\text{def}}{=} \mathsf{len}(A) + 1 \\[1em]
\mathsf{toaddThrds}(S) &\stackrel{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \wedge (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true})\}
\end{aligned}
$$

Figure 53. Auxiliary code to compute the metric for the list traversal.

$A \ ::= \ \epsilon \mid v :: A \qquad tl \ ::= \ \epsilon \mid \mathsf{t} :: tl \qquad L \ ::= \ \epsilon \mid tl + L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadds} * \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadd}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$

$\mathsf{toadds} \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadds}(B) \qquad \mathsf{toadds}(B) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t}} \mathtt{toadd}_{\mathsf{t}} = B(\mathsf{t}))$

$\mathsf{toadd}_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadd}_{\mathsf{t}}(b, B) \qquad \mathsf{toadd}_{\mathsf{t}}(b, B) \stackrel{\text{def}}{=} (\mathtt{toadd}_{\mathsf{t}} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathtt{toadd}_{\mathsf{t}'} = B(\mathsf{t}'))$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', tl, L'.\ A = v :: A' \wedge L = tl + L' \wedge \mathsf{N}_{tl}(x, v, z) * \mathsf{ls}_{L'}(z, A', y))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', tl, L'.\ A = v :: A' \wedge L = tl + L' \wedge \mathsf{N\_irr}_{\mathsf{t},tl}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))$

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

$\mathsf{N}_{tl}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_{tl}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{N\_irr}_{\mathsf{t},tl}(p, v, y) \stackrel{\text{def}}{=} \exists n.\ \mathsf{N\_irr}_{\mathsf{t},tl;n}(p, v, y) \qquad \mathsf{L}_{tl}(p, v, y) \stackrel{\text{def}}{=} \exists n.\ \mathsf{L}_{tl;n}(p, v, y)$

$\mathsf{N\_irr}_{\mathsf{t},tl;n}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L}_{tl;n_1}(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_{tl;n_1}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{lock}_{tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{lock}_{tl;n}(p) \qquad \mathsf{lock}_{tl;n}(p) \stackrel{\text{def}}{=} \mathsf{locked}_{tl;n}(p) \vee (tl = \epsilon \wedge \mathsf{unlocked}_n(p))$

$\mathsf{lock\_irr}_{\mathsf{t},tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) \qquad \mathsf{lock\_irr}_{\mathsf{t},tl;n}(p) \stackrel{\text{def}}{=} \mathsf{lock}_{tl;n}(p) \wedge (\mathsf{t} \notin tl)$

$\mathsf{locked}_{tl}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{locked}_{tl;n}(p) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} \exists n.\ \mathsf{unlocked}_n(p)$

$\mathsf{locked}_{tl;n_1}(p) \stackrel{\text{def}}{=}$
$\quad \exists \mathsf{t}, tl', n_2.\ (tl = \mathsf{t} :: tl') \wedge (\mathsf{t} \notin tl') \wedge ((p.\mathtt{lowner} = n_1) * (p.\mathtt{lnext} = n_2) \wedge (n_1 < n_2))$
$\quad * \mathsf{tickets}(p, 0, n_1) * \mathsf{tickets}_{tl}(p, n_1, n_2) * \mathsf{tickets\_new}(p, n_2)$

$\mathsf{unlocked}_n(p) \stackrel{\text{def}}{=} (p.\mathtt{lowner} = n) * (p.\mathtt{lnext} = n) * \mathsf{tickets}(p, 0, n) * \mathsf{tickets\_new}(p, n)$

$\mathsf{tickets}(p, n_1, n_2) \stackrel{\text{def}}{=} \exists tl.\ \mathsf{tickets}_{tl}(p, n_1, n_2)$

$$\mathsf{tickets}_{tl}(p, n_1, n_2) \stackrel{\text{def}}{=} \begin{cases} (n_1 = n_2) \wedge \mathsf{emp} & \text{if } tl = \epsilon \\ (p.\mathtt{ticket}_{n_1} = \mathsf{t}) * \mathsf{tickets}_{tl'}(p, n_1 + 1, n_2) & \text{if } tl = \mathsf{t} :: tl' \end{cases}$$

$\mathsf{tickets\_new}(p, n_2) \stackrel{\text{def}}{=} (\circledast_{i \geq n_2} p.\mathtt{ticket}_i = -1)$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A)$

$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} = A')$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

$\mathsf{t} \in tl$ iff $\exists \mathsf{t}', tl'.\ (tl = \mathsf{t}' :: tl') \wedge (\mathsf{t} = \mathsf{t}' \vee \mathsf{t} \in tl')$

Figure 54. Invariant and precondition of the lock-coupling list.

$$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathit{ToAdd}_{\mathsf{t}} \vee \mathit{Add}_{\mathsf{t}} \vee \mathit{FailedAdd}_{\mathsf{t}} \vee \mathit{Rmv}_{\mathsf{t}} \vee \mathit{ReqLockH}_{\mathsf{t}} \vee \mathit{ReqLock2}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\mathit{ToAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A, L.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null})] * ((\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}))$$

$$\begin{aligned}\mathit{Add}_{\mathsf{t}} &\stackrel{\text{def}}{=} \\ &\exists x, y, z, n, v, w, u, tl, tl', n_1, S. \\ &((\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) * (\mathtt{S} = S)) \ltimes (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) * \mathsf{U}(y, w, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) * (\mathtt{S} = S \cup \{w\}))) \\ &* [\mathsf{N\_irr}_{\mathsf{t},tl'}(z, u, n) \wedge (v < w < u)]\end{aligned}$$

$$\mathit{FailedAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false})$$

$$\begin{aligned}\mathit{Rmv}_{\mathsf{t}} &\stackrel{\text{def}}{=} \\ &\exists x, y, z, v, u, tl, tl', n_1, S.\ (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) * \mathsf{L}_{\mathsf{t}::tl'}(y, u, z) * (\mathtt{S} = S \uplus \{u\}) \wedge (u < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, z) * (\mathtt{S} = S))\end{aligned}$$

$$\begin{aligned}\mathit{ReqLockH}_{\mathsf{t}} &\stackrel{\text{def}}{=} \\ &\exists x, A, tl, L, n_1.\ (\mathsf{N\_irr}_{\mathsf{t},tl;n_1}(\mathtt{Head}, \mathtt{MIN}, x) \ltimes \mathsf{L}_{tl::\mathsf{t};n_1}(\mathtt{Head}, \mathtt{MIN}, x)) \\ &* [\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, \mathtt{null})]\end{aligned}$$

$$\begin{aligned}\mathit{ReqLock2}_{\mathsf{t}} &\stackrel{\text{def}}{=} \\ &\exists x, y, z, A, v, u, A', L, tl, tl', L', n_1.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}::tl}(x, v, y)] \\ &* (\mathsf{N\_irr}_{\mathsf{t},tl';n_1}(y, u, z) \ltimes \mathsf{L}_{tl'::\mathsf{t};n_1}(y, u, z)) * [(u < \mathtt{MAX}) \wedge \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', \mathtt{null})]\end{aligned}$$

$$\begin{aligned}\mathit{Unlock}_{\mathsf{t}} &\stackrel{\text{def}}{=} \\ &\exists x, y, A, v, L, tl, n_1.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x)] * (\mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) \ltimes \mathsf{N\_irr}_{\mathsf{t},tl;n_1+1}(x, v, y))\end{aligned}$$

$$\mathcal{D}_{\mathsf{t}} \stackrel{\text{def}}{=} \forall x, n_1.\ dp_{\mathsf{t}}(x, n_1) \leadsto dq_{\mathsf{t}}(x, n_1)$$

$$\begin{aligned}dp_{\mathsf{t}}(x, n_1) &\stackrel{\text{def}}{=} \\ &\exists y, z, A, v, u, A', tl, tl', L.\ \mathsf{toadds} * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}::tl;n_1}(x, v, y) \\ &* (\mathsf{U}(y, u, z) \vee \mathsf{L}_{\mathsf{t}::tl'}(y, u, z)) * \mathsf{ls\_unlocked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A')\end{aligned}$$

$$\begin{aligned}dq_{\mathsf{t}}(x, n_1) &\stackrel{\text{def}}{=} \\ &\exists y, z, A, v, u, A', tl, tl', L.\ \mathsf{toadds} * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{N\_irr}_{\mathsf{t},tl;n_1+1}(x, v, y) \\ &* (\mathsf{U}(y, u, z) \vee \mathsf{L}_{\mathsf{t}::tl'}(y, u, z)) * \mathsf{ls\_unlocked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A')\end{aligned}$$

Figure 55. Rely, guarantee and definite actions of the lock-coupling list.

$(\mathit{toadd}, L) < (\mathit{toadd}', L')$ iff one of the following holds:

(1) $\mathsf{tidset}(L) \subset \mathsf{tidset}(L')$, or

(2) $\exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$(\mathsf{toaddThrds}(S, \mathit{toadd}) \subset \mathsf{toaddThrds}(S, \mathit{toadd}'))$, or

(3) $\exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$(\forall \mathsf{t} \in S.\ (\mathit{toadd}(\mathsf{t}) = \mathit{toadd}'(\mathsf{t}))) \wedge$
$(\sum_{\mathsf{t}\in S} \mathsf{backpos}(\mathsf{t}, L) < \sum_{\mathsf{t}\in S} \mathsf{backpos}(\mathsf{t}, L'))$

$(\mathit{toadd}, L) \leq (\mathit{toadd}', L')$ iff
$\exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$(\forall \mathsf{t} \in S.\ (\mathit{toadd}(\mathsf{t}) = \mathit{toadd}'(\mathsf{t}))) \wedge$
$(\sum_{\mathsf{t}\in S} \mathsf{backpos}(\mathsf{t}, L) = \sum_{\mathsf{t}\in S} \mathsf{backpos}(\mathsf{t}, L'))$

$\mathsf{tidset}(\epsilon) \overset{\text{def}}{=} \emptyset \qquad \mathsf{tidset}(\mathit{tl}{+\!\!+}L) \overset{\text{def}}{=} \mathsf{list2set}(\mathit{tl}) \cup \mathsf{tidset}(L)$

$\mathsf{list2set}(\epsilon) \overset{\text{def}}{=} \emptyset \qquad \mathsf{list2set}(\mathsf{t} :: \mathit{tl}) \overset{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{list2set}(\mathit{tl})$

$\mathsf{toaddThrds}(S, \mathit{toadd}) \overset{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \wedge (\mathit{toadd}(\mathsf{t}) = \mathsf{true})\}$

$$\mathsf{backpos}(\mathsf{t}, \epsilon) \overset{\text{def}}{=} 0 \qquad \mathsf{backpos}(\mathsf{t}, \mathit{tl}{+\!\!+}L) \overset{\text{def}}{=} \begin{cases} \mathsf{len}(L) & \text{if } \mathsf{t} \in \mathit{tl} \\ \mathsf{backpos}(\mathsf{t}, L) & \text{if } \mathsf{t} \notin \mathit{tl} \end{cases}$$

$\mathsf{len}(\epsilon) \overset{\text{def}}{=} 0 \qquad \mathsf{len}(\mathit{tl}{+\!\!+}L) \overset{\text{def}}{=} 1 + \mathsf{len}(L)$

$f_1(\mathfrak{S}) = (\mathit{toadd}, L)$ iff $\mathfrak{S} \models J_1(\mathit{toadd}, L)$

$J_1(\mathit{toadd}, L) \overset{\text{def}}{=} \exists z, A, \mathit{tl}, \mathit{tl}', L'.\ (L = \mathit{tl}{+\!\!+}L')$
$\wedge\ \mathsf{toadds}(\mathit{toadd}) * \mathsf{L}_{\mathit{tl}::\mathtt{cid}::\mathit{tl}'}(\mathtt{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L'}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A)$

$J_1 \overset{\text{def}}{=} \exists B, L_0.\ J_1(B, L_0)$

$Q_1 \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_{\mathtt{cid}})$

$f_2(\mathfrak{S}) = (\mathit{toadd}, L)$ iff $\mathfrak{S} \models J_2(\mathit{toadd}, L)$

$J_2(\mathit{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, \mathit{tl}_1, \mathit{tl}, \mathit{tl}', L_2.\ (L = \mathit{tl}{+\!\!+}L_2)$
$\wedge\ \mathsf{toadds}(\mathit{toadd}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, p) * \mathsf{L}_{\mathtt{cid}::\mathit{tl}_1}(p, v, c)$
$* \mathsf{L}_{\mathit{tl}::\mathtt{cid}::\mathit{tl}'}(c, u, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: u :: A_2)$

$J_2 \overset{\text{def}}{=} \exists B, L_0.\ J_2(B, L_0)$

$Q_2 \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_{\mathtt{cid}})$

$f_3(\mathfrak{S}) = (\mathit{toadd}, L)$ iff $\mathfrak{S} \models J_3(\mathit{toadd}, L)$

$J_3(\mathit{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, \mathit{tl}_1, \mathit{tl}, \mathit{tl}', L_2.\ (L = \mathit{tl}{+\!\!+}L_2)$
$\wedge\ \mathsf{toadds}(\mathit{toadd}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, p) * \mathsf{L}_{\mathtt{cid}::\mathit{tl}_1}(p, v, c)$
$* (\mathsf{N\_irr}_{\mathtt{cid},\mathit{tl}}(c, u, z) \vee \mathsf{L}_{\mathit{tl}::\mathtt{cid}::\mathit{tl}'}(c, u, z)) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: u :: A_2)$

$J_3 \overset{\text{def}}{=} \exists B, L_0.\ J_2(B, L_0)$

$Q_3 \overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_{\mathtt{cid}})$

$G_3 \overset{\text{def}}{=} (\mathit{ReqLock2}_{\mathtt{cid}} \vee \mathit{Unlock2}_{\mathtt{cid}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Unlock2}_{\mathsf{t}} \overset{\text{def}}{=}$
$\exists x, y, z, A, v, u, L, \mathit{tl}, \mathit{tl}', n_1.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x)]$
$* (\mathsf{L}_{\mathsf{t}::\mathit{tl};n_1}(x, v, y) \ltimes \mathsf{N\_irr}_{\mathsf{t},\mathit{tl};n_1+1}(x, v, y))\ *\ [\mathsf{L}_{\mathsf{t}::\mathit{tl}'}(y, u, z) \wedge (u < \mathtt{MAX})]$

We can prove:

$J_1 \Rightarrow (R_{\mathtt{cid}}, [I] : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_1} Q_1)$

$J_2 \Rightarrow (R_{\mathtt{cid}}, [I] : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_2} Q_2)$

$J_3 \Rightarrow (R_{\mathtt{cid}}, G_3 : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_3} Q_3)$

Figure 56. The well-founded order and the metrics for the loops.

$$\mathsf{tL}_{tl_1,\mathsf{t},tl_2;n_1,n}(p,v,y) \overset{\text{def}}{=} \mathsf{tlocked}_{tl_1,\mathsf{t},tl_2;n_1,n}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\begin{aligned}&\mathsf{tlocked}_{tl_1,\mathsf{t},tl_2;n_1,n}(p) \overset{\text{def}}{=}\\ &\quad \exists n_2.\ (\mathsf{t} \notin tl_1) \wedge (\mathsf{t} \notin tl_2) \wedge ((p.\mathtt{lowner} = n_1) * (p.\mathtt{lnext} = n_2) \wedge (n_1 \le n < n_2))\\ &\quad * \mathsf{tickets}(p,0,n_1) * \mathsf{tickets}_{tl_1}(p,n_1,n) * \mathsf{tickets}_{\mathsf{t}::tl_2}(p,n,n_2) * \mathsf{tickets\_new}(p,n_2)\end{aligned}$$

$$\begin{aligned}&P_1 \overset{\text{def}}{=}\\ &\quad \exists z, A, tl, tl', L, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{tL}_{tl,\mathtt{cid},tl';n_1,n}(\mathtt{Head},\mathtt{MIN},z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z,A,\mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)\\ &\quad \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{p} = \mathtt{Head}) \wedge (\mathtt{po} \le n_1) \wedge (\mathtt{pi} = n)\end{aligned}$$

$$P_1' \overset{\text{def}}{=} \exists z.\ P_1'(z)$$

$$\begin{aligned}&P_1'(z) \overset{\text{def}}{=}\\ &\quad \exists A, tl, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{Head},\mathtt{MIN},z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z,A,\mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)\\ &\quad \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{p} = \mathtt{Head})\end{aligned}$$

$$\begin{aligned}&P_1'' \overset{\text{def}}{=}\\ &\quad \exists z, A, tl, n_1.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{L}_{\mathtt{cid}::tl;n_1}(\mathtt{Head},\mathtt{MIN},z) * \mathsf{ls\_unlocked}(z,A,\mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)\\ &\quad \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{p} = \mathtt{Head}) \wedge (\mathtt{po} \le n_1) \wedge (\mathtt{pi} = n_1)\end{aligned}$$

$$p_1' \overset{\text{def}}{=} (P_1 \wedge \Diamond(\mathtt{pi} - \mathtt{po}) \vee P_1'' \wedge \Diamond(\mathtt{pi} - (\mathtt{po}+1))) \wedge (\mathtt{pi} \neq \mathtt{po}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$$

```
add(e) {
  local p, c, x, pi, po, ci, co, u, r, aux;
  { $P \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  <p := Head; $\mathtt{toadd}_{\mathtt{cid}}$ := true>;
  { $\exists z, A, tl, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{N\_irr}_{\mathtt{cid},tl}(\mathtt{Head},\mathtt{MIN},z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z,A,\mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)$
    $\wedge (\mathtt{p} = \mathtt{Head}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  <pi := getAndInc(p.lnext); $\mathtt{p.ticket}_{\mathtt{pi}}$ := cid>;
  { $\exists z, A, tl, tl', L, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{tL}_{tl,\mathtt{cid},tl';n_1,n}(\mathtt{Head},\mathtt{MIN},z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z,A,\mathtt{null}) * \mathsf{ss}(\mathtt{MIN}::A)$
    $\wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{p} = \mathtt{Head}) \wedge (\mathtt{pi} = n) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  <po := p.lowner>;
  { $P_1 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  { $P_1 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathtt{pi} - \mathtt{po})$ }
  while (pi <> po) {
    { $p_1'$ }
    <po := p.lowner>;
    { $P_1 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathtt{pi} - \mathtt{po})$ }
  }
  { $P_1' \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  c := p.next;
  { $P_1'(\mathtt{c}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  ...
```

Figure 57. Proof outline for add (1).

$$p_2 \overset{\text{def}}{=} \begin{array}{l} \exists z, A_1, v, A_2, L_1, tl, tl', L_2, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \\ * \, \mathsf{tL}_{tl',\mathtt{cid},\epsilon;n_1,n}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(tl' +\!\!+ L_2))| \leq \mathsf{aux}) \\ \wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{co} \leq n_1) \wedge (\mathtt{ci} = n) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array}$$

$$p_2' \overset{\text{def}}{=} \begin{array}{l} \exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \\ * \, \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \leq \mathsf{aux}) \\ \wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array}$$

$$P_3 \overset{\text{def}}{=} \begin{array}{l} \exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) \\ * \, \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} < \mathtt{MAX}) \\ \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(tl' +\!\!+ L_2))| \leq \mathsf{aux}) \end{array}$$

$$p_3'(c, z) \overset{\text{def}}{=} \begin{array}{l} \exists A_1, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, c) * \mathsf{L}_{\mathtt{cid}::tl}(c, \mathtt{u}, z) * \mathsf{ss}(A_1 :: \mathtt{u} :: A_2) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) \\ \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \leq \mathsf{aux}) \wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array}$$

```
 ...
{ $P_1'(\mathtt{c}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
 <u := c.data;  aux := metric(p)>;
{ $P_3 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
{ $P_3 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathsf{aux})$ }
while (u < e) {
  { $P_3 \wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{u} < \mathtt{e}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  <ci := getAndInc(c.lnext);  $\mathtt{c.ticket}_{\mathtt{ci}}$ := cid>;
  { $\exists z, A_1, v, A_2, L_1, tl, tl', L_2, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2)$
    $* \, \mathsf{tL}_{tl',\mathtt{cid},\epsilon;n_1,n}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(tl' +\!\!+ L_2))| \leq \mathsf{aux})$
    $\wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{ci} = n) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
  <co := c.lowner>;
  { $p_2$ }
  while (ci <> co) {
    <co := c.lowner>;
  }
  { $p_2'$ }
  <p.lowner++>;
  { $\exists z.\ p_3'(\mathtt{c}, z)$ }
  p := c;
  c := p.next;
  { $p_3'(\mathtt{p}, \mathtt{c})$ }
  <u := c.data;  aux := metric(p)>;
  { $P_3 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathsf{aux})$ }
}
{ $P_3 \wedge (\mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
 ...
```

Figure 58. Proof outline for add (2).

$$P_2 \overset{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, tl, tl', L_2, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c})$$
$$* \,\mathsf{tL}_{tl',\mathtt{cid},\epsilon;n_1,n}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$
$$\wedge (\mathtt{co} \le n_1) \wedge (\mathtt{ci} = n) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(tl' +\!\!+ L_2))| \le \mathsf{aux})$$

$$P_2' \overset{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c})$$
$$* \,\mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$
$$\wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \le \mathsf{aux})$$

$$P_2'' \overset{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, tl, n_1.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c})$$
$$* \,\mathsf{L}_{\mathtt{cid};n_1}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_unlocked}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$
$$\wedge (\mathtt{co} \le n_1) \wedge (\mathtt{ci} = n_1) \wedge (1 + \mathsf{len}(A_2) \le \mathsf{aux})$$

```
       { $p_2$ }
       { $P_2 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathtt{ci} - \mathtt{co})$ }
13     while (ci <> co) {
         { $(P_2 \wedge \Diamond(\mathtt{ci} - \mathtt{co}) \vee P_2'' \wedge \Diamond(\mathtt{ci} - (\mathtt{co} + 1))) \wedge (\mathtt{ci} \ne \mathtt{co}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
14       <co := c.lowner>;
         { $P_2 \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \wedge \Diamond(\mathtt{ci} - \mathtt{co})$ }
15     }
       { $P_2' \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
       { $p_2'$ }
```

Figure 59. Proof outline for add (2) – the loop at lines 13-15.

```
     { $P_3 \wedge (\mathtt{e} \le \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
     { $\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c})$
       $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} \le \mathtt{u}) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
21   if (u != e) {
       { $\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c})$
         $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
22     x := cons(0, 0, e, c);
       { $\exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) * \mathsf{U}(\mathtt{x}, \mathtt{e}, \mathtt{c})$
         $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
23     <p.next := x;  toadd_cid := false>;
       { $\exists z, A_1, v, A_2, L_1, tl, tl', tl'', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, \mathtt{x})$
         $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathtt{x}, \mathtt{e}, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},tl''}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: \mathtt{u} :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true})$ }
24     r := true;
25   } else {
26     <r := false;  toadd_cid := false>;
27   }
     { $\exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathtt{p}, v, z)$
       $* \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})$ }
28   <p.lowner++>;
     { $\exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})$ }
     { $P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})$ }
29   return r;
}
```

Figure 60. Proof outline for add (3).

```
rmv(e) {
  local p, c, n, pi, po, ci, co, u, r, aux;
  $\{ P \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
  p := Head;
  ...
  <u := c.data;  aux := metric(p)>;
  while (u < e) {
    <ci := getAndInc(c.lnext);  c.ticket_ci := cid>;
    ...
    <u := c.data;  aux := metric(p)>;
  }
  $\{ \exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, \mathsf{c})$
    $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathsf{c}, \mathsf{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{u} :: A_2) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge (\mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
  if (u = e) {
    $\{ \exists z, A_1, v, A_2, L_1, tl, tl', L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, \mathsf{c})$
      $* \mathsf{N\_irr}_{\mathtt{cid},tl'}(\mathsf{c}, \mathsf{e}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{e} :: A_2) \wedge (\mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
    <ci := getAndInc(c.lnext);  c.ticket_ci := cid>;
    $\{ \exists z, A_1, v, A_2, L_1, tl, tl', L_2, n_1, n.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, \mathsf{c})$
      $* \mathsf{tL}_{tl',\mathtt{cid},\epsilon;n_1,n}(\mathsf{c}, \mathsf{e}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{e} :: A_2) \wedge (\mathsf{e} < \mathtt{MAX}) \wedge (\mathtt{ci} = n) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
    <co := c.lowner>;
    $\{ P_4 \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
    while (ci <> co) {
      <co := c.lowner>;
    }
    $\{ P_4' \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
    n := c.next;
    $\{ \exists A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, \mathsf{c})$
      $* \mathsf{L}_{\mathtt{cid}}(\mathsf{c}, \mathsf{e}, \mathsf{n}) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(\mathsf{n}, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathsf{e} :: A_2) \wedge (\mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathsf{e} = \mathsf{E}) \}$
    p.next := n;
    $\{ \exists A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, \mathsf{n})$
      $* \mathsf{L}_{\mathtt{cid}}(\mathsf{c}, \mathsf{e}, \mathsf{n}) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(\mathsf{n}, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
    <p.lowner++>;
    $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{L}_{\mathtt{cid}}(\mathsf{c}, \mathsf{e}, \mathsf{n}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
    dispose(c);
    $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
    r := true;
  } else {
    $\{ \exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathsf{p}) * \mathsf{L}_{\mathtt{cid}::tl}(\mathsf{p}, v, z)$
      $* \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{false}) \}$
    <p.lowner++>;
    $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{false}) \}$
    r := false;
  }
  $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
  $\{ P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
  return r;
}
```

Figure 61. Proof outline for `rmv`.

$$P_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, tl, tl', L_2, n_1, n.\ \mathsf{toadd}_{\texttt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L_1}(\texttt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\texttt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) \\ * \mathsf{tL}_{tl',\texttt{cid},\epsilon;n_1,n}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_irr}_{\texttt{cid},L_2}(z, A_2, \texttt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2) \wedge (\mathtt{e} < \texttt{MAX}) \wedge (\mathtt{co} \le n_1) \wedge (\mathtt{ci} = n)$$

$$P'_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, tl, L_2.\ \mathsf{toadd}_{\texttt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L_1}(\texttt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\texttt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) \\ * \mathsf{L}_{\texttt{cid}}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_irr}_{\texttt{cid},L_2}(z, A_2, \texttt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2) \wedge (\mathtt{e} < \texttt{MAX})$$

$$P''_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, tl, n_1.\ \mathsf{toadd}_{\texttt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L_1}(\texttt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\texttt{cid}::tl}(\mathtt{p}, v, \mathtt{c}) \\ * \mathsf{L}_{\texttt{cid};n_1}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_unlocked}(z, A_2, \texttt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2) \wedge (\mathtt{e} < \texttt{MAX}) \wedge (\mathtt{co} \le n_1) \wedge (\mathtt{ci} = n_1)$$

$$(P_4 \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E}) \wedge \Diamond(\mathtt{ci} - \mathtt{co}) \wedge (\mathtt{ci} \ne \mathtt{co}) \wedge Q_2 * \mathsf{true}) \\ \Longrightarrow (P''_4 \wedge (\mathtt{ci} \ne \mathtt{co}) \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E}) \wedge \Diamond(\mathtt{ci} - (\mathtt{co} + 1))) * (\Diamond(1) \wedge \mathsf{emp})$$

Also remember $J_2 \Rightarrow (R_{\texttt{cid}}, [I] : \mathcal{D}_{\texttt{cid}} \xrightarrow{f_2} Q_2)$

```
       { $P_4 \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E})$ }
       { $P_4 \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E}) \wedge \Diamond(\mathtt{ci} - \mathtt{co})$ }
24     while (ci <> co) {
         { $(P_4 \wedge \Diamond(\mathtt{ci} - \mathtt{co}) \vee P''_4 \wedge \Diamond(\mathtt{ci} - (\mathtt{co} + 1))) \wedge (\mathtt{ci} \ne \mathtt{co}) \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E})$ }
25       <co := c.lowner>;
         { $P_4 \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E}) \wedge \Diamond(\mathtt{ci} - \mathtt{co})$ }
26     }
       { $P'_4 \wedge \mathsf{arem}(\texttt{RMV}) \wedge (\mathtt{e} = \texttt{E})$ }
```

Figure 62. Proof outline for `rmv` – the loop at lines `24-26`.

D. Deadlock-free examples

In this section, we apply LiLi to verify objects implemented with test-and-set (TAS) locks.

D.1 Counter with TAS lock

The implementation `dfInc` is given in Fig. 1(b). Below we prove that it is linearizable with respect to the atomic operation `INC` and is also starvation-free.

At the left of Fig. 63, we give the specification of the code; and at the right of the figure we show the proof outline. The proofs are similar to the proof of the TAS lock in Sec. 4.3.2, though now we also take into account the $\mathsf{resource}$ of the counters `x` and `X` being protected.

$$I \overset{\text{def}}{=} \mathsf{unlocked} * \mathsf{resource} \vee \mathsf{locked}$$

$$\mathsf{resource} \overset{\text{def}}{=} (\mathtt{x} = \mathtt{X})$$

$$\mathsf{unlocked} \overset{\text{def}}{=} (\mathtt{L} = 0)$$

$$\mathsf{locked} \overset{\text{def}}{=} \exists \mathsf{t}.\ \mathsf{locked}(\mathsf{t})$$

$$\mathsf{locked}(\mathsf{t}) \overset{\text{def}}{=} (\mathtt{L} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS})$$

$$\mathsf{envLocked}(\mathsf{t}) \overset{\text{def}}{=} \exists \mathsf{t}'.\ \mathsf{locked}(\mathsf{t}') \wedge (\mathsf{t}' \neq \mathsf{t})$$

$$P_{\mathtt{cid}} \overset{\text{def}}{=} \mathsf{unlocked} * \mathsf{resource} \vee \mathsf{envLocked}(\mathtt{cid})$$

$$G_{\mathtt{cid}} \overset{\text{def}}{=} (\mathit{Lock}_{\mathtt{cid}} \vee \mathit{Unlock}_{\mathtt{cid}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\mathit{Lock}_{\mathtt{cid}} \overset{\text{def}}{=} (\mathsf{unlocked} * \mathsf{resource}) \ltimes_1 \mathsf{locked}(\mathtt{cid})$$

$$\mathit{Unlock}_{\mathtt{cid}} \overset{\text{def}}{=} \mathsf{locked}(\mathtt{cid}) \ltimes_0 (\mathsf{unlocked} * \mathsf{resource})$$

$$\mathcal{D}_{\mathtt{cid}} \overset{\text{def}}{=} \mathsf{locked}(\mathtt{cid}) \leadsto (\mathsf{unlocked} * \mathsf{resource})$$

$$p \overset{\text{def}}{=} (\mathsf{locked}(\mathtt{cid}) * \mathsf{resource} \wedge \mathtt{b}) \vee (P \wedge (\neg\mathtt{b}) \wedge \blacklozenge(1) \wedge \lozenge(1))$$

$$p' \overset{\text{def}}{=} (\mathsf{unlocked} * \mathsf{resource} \wedge \blacklozenge(1) \wedge \lozenge(0)) \vee (\mathsf{envLocked}(\mathtt{cid}) \wedge \blacklozenge(1) \wedge \lozenge(1))$$

$$Q \overset{\text{def}}{=} \mathsf{locked}(\mathtt{cid}) \vee \mathsf{unlocked} * \mathsf{resource}$$

$$J \overset{\text{def}}{=} \mathsf{locked}(\mathtt{cid}) \vee P$$

```
inc():
  local b, r;
  $\{ P \wedge \blacklozenge(1) \wedge \mathsf{arem}(\mathtt{INC}) \}$
  b := false;
  $\{ p \wedge \mathsf{arem}(\mathtt{INC}) \}$
  while (!b) {
    $\{ p' \wedge \mathsf{arem}(\mathtt{INC}) \}$
    b := cas(&L, 0, cid);
    $\{ p \wedge \mathsf{arem}(\mathtt{INC}) \}$
  }
  $\{ \mathsf{locked}(\mathtt{cid}) * \mathsf{resource} \wedge \mathsf{arem}(\mathtt{INC}) \}$
  r := x; x := r + 1;
  $\{ \mathsf{locked}(\mathtt{cid}) * (\mathtt{x} = \mathtt{X} + 1) \wedge \mathsf{arem}(\mathtt{INC}) \}$
  L := 0;
  $\{ P \wedge \mathsf{arem}(\mathbf{skip}) \}$
```

Figure 63. Proof outline of the counter with TAS lock.

D.2 Two-lock queue with TAS lock

Below we prove that the two-lock queue implemented with TAS locks is deadlock-free. The implementation is shown in Fig. 64.

The proofs are similar to the proofs in Sec. C.5 where the locks are implemented using ticket locks. Fig. 65 defines the precise invariant, the precondition, the rely and guarantee conditions and the definite action. Fig. 66 and Fig. 67 show the proof outlines. Note that the proofs for the loops of acquiring the locks follow the proofs for the counter with TAS locks (see Sec. D.1).

```
enq(v) {
  local x, b;
  x := cons(v, null);
  b := false;
  while (!b) {
    b := cas(&Tlock, 0, cid);
  }
  Tail.next := x;
  Tail := x;
  Tlock := 0;
}
```

```
int deq() {
  local h, s, v, b;
  b := false;
  while (!b) {
    b := cas(&Hlock, 0, cid);
  }
  h := Head;
  s := h.next;
  if (s = null) {
    Hlock := 0;
    v := EMPTY;
  } else {
    v := s.data;
    Head := s;
    Hlock := 0;
    dispose(h);
  }
  return v;
}
```

Figure 64. Two-lock queue implementation with TAS lock.

$I \overset{\text{def}}{=} \exists h, z, s.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(s, h, z) * \mathsf{lock}(\texttt{Hlock}) * \mathsf{lock}_s(\texttt{Tlock})$

$\mathsf{queue}(s, h, z) \overset{\text{def}}{=} \exists v_d, A.\ (\texttt{Q} = A) * (\mathsf{unlag}(h, z, v_d :: A) \vee (\mathsf{lag}(h, z, \_, v_d :: A) \wedge (s \neq 0)) \vee (\mathsf{cross}(h, v_d :: A) \wedge (s \neq 0)))$

$\mathsf{unlag}(h, z, A) \overset{\text{def}}{=} \exists v, A'.\ (A = A' :: v) \wedge \mathsf{ls}(h, A', z) * \mathsf{N}(z, v, \texttt{null})$

$\mathsf{lag}(h, z, x, A) \overset{\text{def}}{=} \exists v, v', A'.\ (A = A' :: v :: v') \wedge \mathsf{ls}(h, A', z) * \mathsf{N2}(z, v, x, v', \texttt{null})$

$\mathsf{cross}(h, A) \overset{\text{def}}{=} \exists v.\ (A = v :: \epsilon) \wedge \mathsf{N}(h, v, \texttt{null})$

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

$\mathsf{N}(p, v, y) \overset{\text{def}}{=} (p.\texttt{data} = v) * (p.\texttt{next} = y)$ $\qquad$ $\mathsf{N2}(p, v, y, v', z) \overset{\text{def}}{=} \mathsf{N}(p, v, y) * \mathsf{N}(y, v', z)$

$\mathsf{lock}(l) \overset{\text{def}}{=} \exists s.\ \mathsf{lock}_s(l)$ $\qquad$ $\mathsf{lock}_s(l) \overset{\text{def}}{=} (l = s)$ $\qquad$ $\mathsf{lock\_irr}_{\mathsf{t}}(l) \overset{\text{def}}{=} \exists s.\ \mathsf{lock\_irr}_{\mathsf{t},s}(l)$ $\qquad$ $\mathsf{lock\_irr}_{\mathsf{t},s}(l) \overset{\text{def}}{=} \mathsf{lock}_s(l) \wedge (s \neq \mathsf{t})$

$\mathsf{unlocked}(l) \overset{\text{def}}{=} \mathsf{lock}_0(l)$ $\qquad$ $\mathsf{locked}(l) \overset{\text{def}}{=} \exists \mathsf{t}.\ \mathsf{locked}_{\mathsf{t}}(l)$ $\qquad$ $\mathsf{locked}_{\mathsf{t}}(l) \overset{\text{def}}{=} \mathsf{lock}_{\mathsf{t}}(l) \wedge (\mathsf{t} \in \texttt{TIDS})$

$P_{\mathsf{t}} \overset{\text{def}}{=} \exists h, z, s.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(s, h, z) * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t},s}(\texttt{Tlock})$

$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \overset{\text{def}}{=} (\mathit{Enq}_{\mathsf{t}} \vee \mathit{Swing}_{\mathsf{t}} \vee \mathit{Deq}_{\mathsf{t}} \vee \mathit{LockH}_{\mathsf{t}} \vee \mathit{UnlockH}_{\mathsf{t}} \vee \mathit{LockT}_{\mathsf{t}} \vee \mathit{UnlockT}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Enq}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, A, v, v'.\ [\mathsf{locked}_{\mathsf{t}}(\texttt{Tlock}) * (\texttt{Tail} = x)] * ((\mathsf{N}(x, v, \texttt{null}) * (\texttt{Q} = A)) \ltimes_0 (\mathsf{N2}(x, v, y, v', \texttt{null}) * (\texttt{Q} = A :: v')))$

$\mathit{Swing}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, v.\ [\mathsf{locked}_{\mathsf{t}}(\texttt{Tlock}) * \mathsf{N}(x, v, \texttt{null})] * ((\texttt{Tail} = \_) \ltimes_0 (\texttt{Tail} = x))$

$\mathit{Deq}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, z, v, v', A.\ [\mathsf{locked}_{\mathsf{t}}(\texttt{Hlock})] * (((\texttt{Head} = x) * \mathsf{N2}(x, v, y, v', z) * (\texttt{Q} = v' :: A)) \ltimes_0 ((\texttt{Head} = y) * \mathsf{N}(y, v', z) * (\texttt{Q} = A)))$

$\mathit{LockH}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad [\mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Tlock})] * (\mathsf{unlocked}(\texttt{Hlock}) \ltimes_1 \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}))$

$\mathit{UnlockH}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad [\mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Tlock})] * (\mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) \ltimes_0 \mathsf{unlocked}(\texttt{Hlock}))$

$\mathit{LockT}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad [\mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock})] * (\mathsf{unlocked}(\texttt{Tlock}) \ltimes_1 \mathsf{locked}_{\mathsf{t}}(\texttt{Tlock}))$

$\mathit{UnlockT}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists z, v.\ [\mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * (\texttt{Tail} = z) * \mathsf{N}(z, v, \texttt{null})] * (\mathsf{locked}_{\mathsf{t}}(\texttt{Tlock}) \ltimes_0 \mathsf{unlocked}(\texttt{Tlock}))$

$\mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} (dpH_{\mathsf{t}} \rightsquigarrow dqH_{\mathsf{t}}) \wedge (dpT_{\mathsf{t}} \rightsquigarrow dqT_{\mathsf{t}})$

$dpH_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{true} \wedge I$ $\qquad$ $dqH_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{unlocked}(\texttt{Hlock}) * \mathsf{true} \wedge I$

$dpT_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{locked}_{\mathsf{t}}(\texttt{Tlock}) * \mathsf{true} \wedge I$ $\qquad$ $dqT_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{unlocked}(\texttt{Tlock}) * \mathsf{true} \wedge I$

Figure 65. Invariant, precondition, rely/guarantee and definite actions of the two-lock queue with TAS locks.

$$P_1 \overset{\text{def}}{=} \exists h, z, v_d, A.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * \mathsf{unlag}(h, z, v_d :: A) * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{locked}_{\mathsf{t}}(\texttt{Tlock})$$

$$P_2(x) \overset{\text{def}}{=} \exists h, z, v_d, A.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * (\mathsf{lag}(h, z, x, v_d :: A) \vee (\mathsf{cross}(h, v_d :: A) \wedge (h = x))) * \mathsf{lock\_irr}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{locked}_{\mathsf{t}}(\texttt{Tlock})$$

```
enq(v) {
  local x, b;
  $\{\, P \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \blacklozenge(1) \,\}$
  x := cons(v, null);
  $\{\, P * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \wedge \blacklozenge(1) \,\}$
  b := false;
  $\{\, ((\neg\texttt{b} \wedge P \wedge \blacklozenge(1) \wedge \Diamond(1)) \vee (\texttt{b} \wedge P_1)) * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \,\}$
  while (!b) {
    b := cas(&Tlock, 0, cid);
  }
  $\{\, P_1 * \mathsf{N}(\texttt{x}, \texttt{v}, \texttt{null}) \wedge \mathsf{arem}(\texttt{ENQ}) \wedge (\texttt{v} = \texttt{V}) \,\}$
  Tail.next := x;
  $\{\, P_2(\texttt{x}) \wedge \mathsf{arem}(\textbf{skip}) \,\}$
  Tail := x;
  $\{\, P_1 \wedge \mathsf{arem}(\textbf{skip}) \,\}$
  Tlock := 0;
  $\{\, P \wedge \mathsf{arem}(\textbf{skip}) \,\}$
}
```

Figure 66. Proof outline for enq.

$$P'_1(h) \overset{\text{def}}{=} \exists z, s.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * \mathsf{queue}(s, h, z) * \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t},s}(\texttt{Tlock})$$

$$P'_2(h, x) \overset{\text{def}}{=} \exists z, A, s.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = A) * \mathsf{N}(h, \_, x) * (\mathsf{unlag}(x, z, A) \vee \mathsf{lag}(x, z, \_, A) \wedge (s \neq 0)) * \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t},s}(\texttt{Tlock})$$

$$P'_3(h, x, v) \overset{\text{def}}{=} \exists s.\ (\texttt{Head} = h) * (\texttt{Tail} = h) * (\texttt{Q} = v :: \epsilon) * \mathsf{N2}(h, \_, x, v, \texttt{null}) * \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t},s}(\texttt{Tlock}) \wedge (s \neq 0)$$

$$\begin{aligned} P'_4(h, x, v) \overset{\text{def}}{=}\ & \exists y, z, v, A, s.\ (\texttt{Head} = h) * (\texttt{Tail} = z) * (\texttt{Q} = v :: A) * \mathsf{N2}(h, \_, x, v, y) * \mathsf{locked}_{\mathsf{t}}(\texttt{Hlock}) * \mathsf{lock\_irr}_{\mathsf{t},s}(\texttt{Tlock}) \\ & * ((x = z) \wedge (y = \texttt{null}) \wedge (A = \epsilon) \vee \mathsf{unlag}(y, z, A)\ \vee\ (x = z) \wedge \mathsf{N}(y, v', \texttt{null}) \wedge (A = v' :: \epsilon) \wedge (s \neq 0) \vee \mathsf{lag}(y, z, \_, A) \wedge (s \neq 0)) \end{aligned}$$

```
int deq() {
  local h, s, v, b;
  $\{\, P \wedge \mathsf{arem}(\texttt{DEQ}) \wedge \blacklozenge(1) \,\}$
  b := false;
  $\{\, ((\neg\texttt{b} \wedge P \wedge \blacklozenge(1) \wedge \Diamond(1)) \vee (\exists h.\ \texttt{b} \wedge P'_1(h))) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
  while (!b) {
    b := cas(&Hlock, 0, cid);
  }
  $\{\, \exists h.\ P'_1(h) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
  h := Head;
  $\{\, P'_1(\texttt{h}) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
  s := h.next;
  $\{\, ((\texttt{s} = \texttt{null}) \wedge P'_1(\texttt{h}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY})) \vee (P'_2(\texttt{h}, \texttt{s}) \vee P'_3(\texttt{h}, \texttt{s}, \_)) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
  if (s = null) {
    $\{\, P'_1(\texttt{h}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY}) \,\}$
    Hlock := 0;
    $\{\, P \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{V} = \texttt{EMPTY}) \,\}$
    v := EMPTY;
  } else {
    $\{\, (P'_2(\texttt{h}, \texttt{s}) \vee P'_3(\texttt{h}, \texttt{s}, \_)) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
    v := s.data;
    $\{\, (P'_4(\texttt{h}, \texttt{s}, \texttt{v}) \vee P'_3(\texttt{h}, \texttt{s}, \texttt{v})) \wedge \mathsf{arem}(\texttt{DEQ}) \,\}$
    Head := s;
    $\{\, P'_1(\texttt{s}) * \mathsf{N}(\texttt{h}, \_, \texttt{s}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V}) \,\}$
    Hlock := 0;
    $\{\, P * \mathsf{N}(\texttt{h}, \_, \texttt{s}) \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V}) \,\}$
    dispose(h);
  }
  $\{\, P \wedge \mathsf{arem}(\textbf{skip}) \wedge (\texttt{v} = \texttt{V}) \,\}$
  return v;
}
```

Figure 67. Proof outline for deq.

D.3 Lock-coupling list with TAS lock

Fig. 68 shows the code of the lock-coupling list implemented with TAS locks (where the auxiliary code for computing the metric of the list traversal is shown in Fig. 69). We prove its deadlock-freedom.

Fig. 70 defines the precise invariant and the precondition. Fig. 71 defines the rely/guarantee conditions and the definite actions. These definitions are similar to the ones for verifying the lock-coupling list with ticket locks (Appendix C.6). Other figures in this section give the proof outlines.

In the proofs, the only delaying action of a thread t is that t acquires the lock of the *head* node in the list. Thus each method is given one ⧫-token only. After acquiring the lock of the head node, the threads doing the operations stand in line. As a result, the list implementation will become starvation-free so long as the lock of the head node is implemented using a ticket lock or a queue lock (even if the locks of other nodes are implemented using TAS locks).

```
bool[] toadd; //initially all false

struct Node {
  int lock;
  int data;
  struct Node *next;
}
```

```
struct List {
  struct Node *Head;
}

initialize(){
  Head := cons(0, MIN, null);
  Head.next := cons(0, MAX, null);
}
```

```
add(e) {
  local p, c, x, b, u, r, aux;
  <p := Head; toadd_cid := true>;
  b := false;
  while (!b) {
    b := cas(p.lock, 0, cid);
  }
  c := p.next;
  <u := c.data; aux := metricTAS(p)>;
  while (u < e) {
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    p.lock := 0;
    p := c;
    c := p.next;
    <u := c.data; aux := metricTAS(p)>;
  }
  if (u != e) {
    x := cons(0, e, c);
    <p.next := x; toadd_cid := false>;
    r := true;
  } else {
    <r := false; toadd_cid := false>;
  }
  p.lock := 0;
  return r;
}
```

```
rmv(e) {
  local p, c, n, b, u, r, aux;
  p := Head;
  b := false;
  while (!b) {
    b := cas(p.lock, 0, cid);
  }
  c := p.next;
  <u := c.data; aux := metricTAS(p)>;
  while (u < e) {
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    p.lock := 0;
    p := c;
    c := p.next;
    <u := c.data; aux := metricTAS(p)>;
  }
  if (u = e) {
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    n := c.next;
    p.next := n;
    p.lock := 0;
    dispose(c);
    r := true;
  } else {
    p.lock := 0;
    r := false;
  }
  return r;
}
```

Figure 68. Lock-coupling list with TAS lock.

```
int metricTAS(p){
  local n, l, ts, o, t;
  n := p.next; l := 0; ts := ∅;
  while (n <> null) {
    l++;
    t := n.lock;
    if (t != 0) {
      if (toadd_t = true) ts := ts ∪ {t};
    }
    n := n.next;
  }
  l := l + |ts|;
  return l;
}
```

$$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1$$

$$\mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$$

$$\mathsf{toaddThrds}(S) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \land (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true})\}$$

Figure 69. Auxiliary code to compute the metric for the list traversal.

$$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$$

$$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadds} * \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$$

$$P_{\mathtt{cid}} \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A)$$

$$\mathsf{toadds} \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadds}(B) \qquad \mathsf{toadds}(B) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t}} \mathtt{toadd}_{\mathsf{t}} = B(\mathsf{t}))$$

$$\mathsf{toadd}_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists B.\ \mathsf{toadd}_{\mathsf{t}}(b, B) \qquad \mathsf{toadd}_{\mathsf{t}}(b, B) \stackrel{\text{def}}{=} (\mathtt{toadd}_{\mathsf{t}} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathtt{toadd}_{\mathsf{t}'} = B(\mathsf{t}'))$$

$$\begin{aligned}
&\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=} \\
&\qquad (x = y \land A = \epsilon \land L = \epsilon \land \mathsf{emp}) \\
&\qquad \lor (x \neq y \land \exists z, v, A', s, L'.\ A = v :: A' \land L = s :: L' \land \mathsf{N}_s(x, v, z) * \mathsf{ls}_{L'}(z, A', y))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=} \\
&\qquad (x = y \land A = \epsilon \land L = \epsilon \land \mathsf{emp}) \\
&\qquad \lor (x \neq y \land \exists z, v, A', s, L'.\ A = v :: A' \land L = s :: L' \land \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ls\_unlocked}(x, A, y) \stackrel{\text{def}}{=} \\
&\qquad (x = y \land A = \epsilon \land \mathsf{emp}) \ \lor\ (x \neq y \land \exists z, v, A'.\ A = v :: A' \land \mathsf{U}(x, v, z) * \mathsf{ls\_unlocked}(z, A', y))
\end{aligned}$$

$$\mathsf{N}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{L}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \land \mathsf{L}_s(p, v, y)$$

$$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$$

$$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \lor (s = 0 \land \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \land (\mathsf{t} \neq s)$$

$$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \land (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$$

$$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \land \mathsf{sorted}(A)$$

$$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \land \mathsf{sorted}(A) \land (\mathtt{S} = A')$$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \lor A = v :: \epsilon \\ (v_1 < v_2) \land \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

Figure 70. Invariant and precondition of the lock-coupling list with TAS lock.

$G_{\mathtt{cid}} \stackrel{\text{def}}{=} (\mathit{ToAdd}_{\mathtt{cid}} \vee \mathit{Add}_{\mathtt{cid}} \vee \mathit{FailedAdd}_{\mathtt{cid}} \vee \mathit{Rmv}_{\mathtt{cid}} \vee \mathit{LockH}_{\mathtt{cid}} \vee \mathit{Lock2}_{\mathtt{cid}} \vee \mathit{Unlock}_{\mathtt{cid}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{ToAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A, L.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null})] * ((\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}))$

$\mathit{Add}_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists x, y, z, n, v, w, u, s, S.$
$\quad ((\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) * (\mathtt{S} = S)) \ltimes (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{U}(y, w, z) * (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false}) * (\mathtt{S} = S \cup \{w\})))$
$\quad * [\mathsf{N\_irr}_{\mathsf{t},s}(z, u, n) \wedge (v < w < u)]$

$\mathit{FailedAdd}_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathtt{toadd}_{\mathsf{t}} = \mathsf{true}) \ltimes (\mathtt{toadd}_{\mathsf{t}} = \mathsf{false})$

$\mathit{Rmv}_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists x, y, z, v, u, S.\ (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{L}_{\mathsf{t}}(y, u, z) * (\mathtt{S} = S \uplus \{u\}) \wedge (u < \mathtt{MAX})) \ltimes (\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathtt{S} = S))$

$\mathit{LockH}_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists x, A, L.\ (\mathsf{U}(\mathtt{Head}, \mathtt{MIN}, x) \ltimes_1 \mathsf{L}_{\mathsf{t}}(\mathtt{Head}, \mathtt{MIN}, x)) * [\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, \mathtt{null})]$

$\mathit{Lock2}_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists x, y, z, A, v, u, A', L, L'.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y)] * (\mathsf{U}(y, u, z) \ltimes \mathsf{L}_{\mathsf{t}}(y, u, z))$
$\quad * [(u < \mathtt{MAX}) \wedge \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', \mathtt{null})]$

$\mathit{Unlock}_{\mathsf{t}} \stackrel{\text{def}}{=}$
$\quad \exists x, y, A, v, L.\ [\mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x)] * (\mathsf{L}_{\mathsf{t}}(x, v, y) \ltimes \mathsf{U}(x, v, y))$

$\mathcal{D}_{\mathtt{cid}} \stackrel{\text{def}}{=} \forall x.\ dp_{\mathtt{cid}}(x) \leadsto dq_{\mathtt{cid}}(x)$

$dp_{\mathsf{t}}(x) \stackrel{\text{def}}{=}$
$\quad \exists y, z, A, v, u, A', L.\ \mathsf{toadds} * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y)$
$\quad * (\mathsf{U}(y, u, z) \vee \mathsf{L}_{\mathsf{t}}(y, u, z)) * \mathsf{ls\_unlocked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A')$

$dq_{\mathsf{t}}(x) \stackrel{\text{def}}{=}$
$\quad \exists y, z, A, v, u, A', L.\ \mathsf{toadds} * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{U}(x, v, y)$
$\quad * (\mathsf{U}(y, u, z) \vee \mathsf{L}_{\mathsf{t}}(y, u, z)) * \mathsf{ls\_unlocked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A')$

Figure 71. Rely, guarantee and definite actions of the lock-coupling list with TAS lock.

$(\mathit{toadd}, L) < (\mathit{toadd}', L')$ iff one of the following holds:

(1) $\mathsf{tidset}(L) \subset \mathsf{tidset}(L')$, or

(2) $\exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$(\mathsf{toaddThrds}(S, \mathit{toadd}) \subset \mathsf{toaddThrds}(S, \mathit{toadd}'))$, or

(3) $\exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$(\forall \mathsf{t} \in S.\ (\mathit{toadd}(\mathsf{t}) = \mathit{toadd}'(\mathsf{t}))) \wedge$
$(\sum_{\mathsf{t} \in S} \mathsf{backpos}(\mathsf{t}, L) < \sum_{\mathsf{t} \in S} \mathsf{backpos}(\mathsf{t}, L'))$

$(\mathit{toadd}, L) \leq (\mathit{toadd}', L')$ iff
$\quad \exists S.\ (\mathsf{tidset}(L) = \mathsf{tidset}(L') = S) \wedge$
$\quad (\forall \mathsf{t} \in S.\ (\mathit{toadd}(\mathsf{t}) = \mathit{toadd}'(\mathsf{t}))) \wedge$
$\quad (\sum_{\mathsf{t} \in S} \mathsf{backpos}(\mathsf{t}, L) = \sum_{\mathsf{t} \in S} \mathsf{backpos}(\mathsf{t}, L'))$

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

$\mathsf{tidset}(\mathsf{t} :: L) \stackrel{\text{def}}{=} \{\mathsf{t}\} \cup \mathsf{tidset}(L)$

$\mathsf{tidset}((0) :: L) \stackrel{\text{def}}{=} \mathsf{tidset}(L)$

$\mathsf{toaddThrds}(S, \mathit{toadd}) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\mathsf{t} \in S) \wedge (\mathit{toadd}(\mathsf{t}) = \mathsf{true})\}$

$\mathsf{backpos}(\mathsf{t}, \epsilon) \stackrel{\text{def}}{=} 0$

$$\mathsf{backpos}(\mathsf{t}, s :: L) \stackrel{\text{def}}{=} \begin{cases} \mathsf{len}(L) & \text{if } \mathsf{t} = s \\ \mathsf{backpos}(\mathsf{t}, L) & \text{if } \mathsf{t} \neq s \end{cases}$$

$\mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0$

$\mathsf{len}(s :: L) \stackrel{\text{def}}{=} 1 + \mathsf{len}(L)$

Figure 72. The well-founded order.

$f_1(\mathfrak{S}) = (\mathit{toadd}, L)$ iff $\mathfrak{S} \models J_1(\mathit{toadd}, L)$

$J_1(\mathit{toadd}, L) \overset{\text{def}}{=} \exists z, A, s, L'.\ (L = s :: L')$
$\quad \wedge\ \mathsf{toadds}(\mathit{toadd}) * \mathsf{N}_s(\mathtt{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L'}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A)$

$J_1 \overset{\text{def}}{=} \exists B, L_0.\ J_1(B, L_0)$

$Q_1 \overset{\text{def}}{=} \exists z, A.\ \mathsf{toadds} * (\mathsf{L}_{\mathtt{cid}}(\mathtt{Head}, \mathtt{MIN}, z) \vee \mathsf{U}(\mathtt{Head}, \mathtt{MIN}, z)) * \mathsf{ls\_unlocked}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A)$

$f_2(\mathfrak{S}) = (\mathit{toadd}, L)$ iff $\mathfrak{S} \models J_2(\mathit{toadd}, L)$

$J_2(\mathit{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, s, L_2.\ (L = s :: L_2)$
$\quad \wedge\ \mathsf{toadds}(\mathit{toadd}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, p) * \mathsf{L}_{\mathtt{cid}}(p, v, c)$
$\quad * \mathsf{N}_s(c, u, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: u :: A_2)$

$J_2 \overset{\text{def}}{=} \exists B, L_0.\ J_2(B, L_0)$

$Q_2 \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1.\ \mathsf{toadds} * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, p) * \mathsf{L}_{\mathtt{cid}}(p, v, c)$
$\quad * (\mathsf{U}(c, u, z) \vee \mathsf{L}_{\mathtt{cid}}(c, u, z)) * \mathsf{ls\_unlocked}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: u :: A_2)$

$G_3 \overset{\text{def}}{=} (\mathit{Lock2}_{\mathtt{cid}} \vee \mathit{Unlock2}_{\mathtt{cid}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Unlock2}_{\mathtt{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, z, A, v, u, L.\ [\mathsf{ls\_irr}_{\mathtt{t},L}(\mathtt{Head}, A, x)] * (\mathsf{L}_{\mathtt{t}}(x, v, y) \ltimes \mathsf{U}(x, v, y)) * [\mathsf{L}_{\mathtt{t}}(y, u, z) \wedge (u < \mathtt{MAX})]$

We can prove:

$J_1 \Rightarrow (R_{\mathtt{cid}}, [I] : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_1} Q_1)$
$J_2 \Rightarrow (R_{\mathtt{cid}}, [I] : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_2} Q_2)$
$J_2 \Rightarrow (R_{\mathtt{cid}}, G_3 : \mathcal{D}_{\mathtt{cid}} \xrightarrow{f_2} Q_2)$

Figure 73. The metrics.

$p_1 \overset{\text{def}}{=} \exists z, A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{U}(\mathtt{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A) \wedge (\mathtt{p} = \mathtt{Head})$

$p_1' \overset{\text{def}}{=} \exists z, A, s, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{L\_irr}_{\mathtt{cid},s}(\mathtt{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A) \wedge (\mathtt{p} = \mathtt{Head})$

$p_1'' \overset{\text{def}}{=} \exists z.\ p_1'(z)$

$p_1''(z) \overset{\text{def}}{=} \exists A, s, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{Head}, \mathtt{MIN}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A, \mathtt{null}) * \mathsf{ss}(\mathtt{MIN} :: A) \wedge (\mathtt{p} = \mathtt{Head})$

```
add(e) {
 1  local p, c, x, b, u, r, aux;
    $\{ P \wedge \blacklozenge(1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 2  <p := Head; $\mathtt{toadd}_{\mathtt{cid}}$ := true>;
    $\{ (p_1 \vee p_1') \wedge \blacklozenge(1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 3  b := false;
    $\{ ((p_1'' \wedge \mathtt{b}) \vee (p_1 \vee p_1') \wedge \neg\mathtt{b} \wedge \blacklozenge(1) \wedge \lozenge(1)) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 4  while (!b) {
      $\{ ((p_1 \wedge \blacklozenge(1) \wedge \lozenge(0)) \vee (p_1' \wedge \blacklozenge(1) \wedge \lozenge(1))) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 5    b := cas(p.lock, 0, cid);
      $\{ ((p_1'' \wedge \mathtt{b}) \vee (p_1 \vee p_1') \wedge \neg\mathtt{b} \wedge \blacklozenge(1) \wedge \lozenge(1)) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 6  }
    $\{ p_1'' \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 7  c := p.next;
    $\{ p_1''(\mathtt{c}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
 8  ...
```

Figure 74. Proof outline for add (1).

$$p_2 \stackrel{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, s, L_2.\, \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null})$$
$$* \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(s :: L_2))| \leq \mathsf{aux}) \wedge (v < \mathtt{e} < \mathtt{MAX})$$

$$p_2''' \stackrel{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, L_2.\, \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null})$$
$$* \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \leq \mathsf{aux}) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$

$$p_3'''(c, z) \stackrel{\text{def}}{=}$$
$$\exists A_1, A_2, L_1, L_2.\, \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, c) * \mathsf{L}_{\mathtt{cid}}(c, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null})$$
$$* \mathsf{ss}(A_1 :: \mathtt{u} :: A_2) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(L_2))| \leq \mathsf{aux}) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$

```
7   ...
   { $p_1''(\mathtt{c}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E})$ }
8   <u := c.data;  aux := metricTAS(p)>;
   { $p_2 \wedge \Diamond(\mathsf{aux}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
9   while (u < e) {
     { $p_2 \wedge \Diamond(\mathsf{aux} - 1) \wedge (\mathtt{u} < \mathtt{e}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
10    b := false;
     { $(p_2''' \wedge \mathtt{b} \vee p_2 \wedge (\mathtt{u} < \mathtt{e}) \wedge \neg\mathtt{b}) \wedge \Diamond(\mathsf{aux} - 1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
11    while (!b) {
12      b := cas(c.lock, 0, cid);
13    }
     { $p_2''' \wedge \Diamond(\mathsf{aux} - 1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
14    p.lock := 0;
     { $\exists z.\, p_3'''(\mathtt{c}, z) \wedge \Diamond(\mathsf{aux} - 1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
15    p := c;
16    c := p.next;
     { $p_3'''(\mathtt{p}, \mathtt{c}) \wedge \Diamond(\mathsf{aux} - 1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
17    <u := c.data;  aux := metricTAS(p)>;
     { $p_2 \wedge \Diamond(\mathsf{aux}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
18  }
   { $p_2 \wedge (\mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})$ }
19  ...
```

Figure 75. Proof outline for `add` (2).

$$p_2' \stackrel{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, L_2.\, \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{U}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null})$$
$$* \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(s :: L_2))| \leq \mathsf{aux}) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$

$$p_2'' \stackrel{\text{def}}{=}$$
$$\exists z, A_1, v, A_2, L_1, s, L_2.\, \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{L\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null})$$
$$* \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (1 + \mathsf{len}(A_2) + |\mathsf{toaddThrds}(\mathsf{tidset}(s :: L_2))| \leq \mathsf{aux}) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX})$$

```
     { $(p_2''' \wedge \mathtt{b}) \vee (p_2 \wedge (\mathtt{u} < \mathtt{e}) \wedge \neg\mathtt{b})$ }
     { $(p_2''' \wedge \mathtt{b}) \vee (p_2 \wedge (\mathtt{u} < \mathtt{e}) \wedge \neg\mathtt{b} \wedge \Diamond(1))$ }
11    while (!b) {
       { $(p_2' \wedge \Diamond(0)) \vee (p_2'' \wedge \Diamond(1))$ }
12      b := cas(c.lock, 0, cid);
       { $(p_2''' \wedge \mathtt{b}) \vee (p_2 \wedge (\mathtt{u} < \mathtt{e}) \wedge \neg\mathtt{b} \wedge \Diamond(1))$ }
13    }
     { $p_2'''$ }
```

Figure 76. Proof outline for `add` (2) – the loop at lines `11-13`.

```
$\{ p_2 \wedge (\mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
$\left\{ \begin{array}{l} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) \\ \quad * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array} \right\}$
if (u != e) {
  $\left\{ \begin{array}{l} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) \\ \quad * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array} \right\}$
  x := cons(0, e, c);
  $\left\{ \begin{array}{l} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{U}(\mathtt{x}, \mathtt{e}, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) \\ \quad * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array} \right\}$
  <p.next := x;  $\mathtt{toadd}_{\mathtt{cid}}$ := false>;
  $\left\{ \begin{array}{l} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{x}) * \mathsf{U}(\mathtt{x}, \mathtt{e}, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) \\ \quad * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: \mathtt{u} :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \end{array} \right\}$
  r := true;
} else {
  <r := false;  $\mathtt{toadd}_{\mathtt{cid}}$ := false>;
}
$\left\{ \begin{array}{l} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, z) \\ \quad * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \end{array} \right\}$
p.lock := 0;
$\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
$\{ P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
return r;
}
```

Figure 77. Proof outline for `add` (3).

$$p_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{U}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2)$$

$$p'_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{L\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2)$$

$$p''_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{e}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2)$$

```
rmv(e) {
1   local p, c, n, pi, po, ci, co, u, r, aux;
    $\{ P \wedge \blacklozenge(1) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
2   p := Head;
.   ...
8   <u := c.data;  aux := metricTAS(p)>;
9   while (u < e) {
..    ...
18  }
    $\{ \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z)$
    $\ * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
19  if (u = e) {
      $\{ \exists z, A_1, v, A_2, L_1, s, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{e}, z)$
      $\ * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
20    b := false;
      $\{ ((p''_4 \wedge \mathtt{b}) \vee (p_4 \vee p'_4) \wedge \neg\mathtt{b} \wedge \lozenge(1)) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
21    while (!b) {
        $\{ ((p_4 \wedge \lozenge(0)) \vee (p'_4 \wedge \lozenge(1))) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
22      b := cas(c.lock, 0, cid);
        $\{ ((p''_4 \wedge \mathtt{b}) \vee (p_4 \vee p'_4) \wedge \neg\mathtt{b} \wedge \lozenge(1)) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
23    }
      $\{ p''_4 \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
24    n := c.next;
      $\{ \exists A_1, v, A_2, L_1, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{c}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{e}, \mathtt{n})$
      $\ * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(\mathtt{n}, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{e} :: A_2) \wedge (\mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E}) \}$
25    p.next := n;
      $\{ \exists A_1, v, A_2, L_1, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, \mathtt{n}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{e}, \mathtt{n})$
      $\ * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(\mathtt{n}, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
26    p.lock := 0;
      $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{c}, \mathtt{e}, \mathtt{n}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
27    dispose(c);
      $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true}) \}$
28    r := true;
29  } else {
      $\{ \exists z, A_1, v, A_2, L_1, L_2.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{L}_{\mathtt{cid}}(\mathtt{p}, v, z)$
      $\ * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: A_2) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{false}) \}$
30    p.lock := 0;
      $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{false}) \}$
31    r := false;
32  }
    $\{ \exists A, L.\ \mathsf{toadd}_{\mathtt{cid}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
    $\{ P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R}) \}$
33  return r;
}
```

Figure 78. Proof outline for `rmv`.

D.4 Optimistic list with TAS lock

Fig. 79 shows the code of the optimistic list implemented with TAS locks (where the auxiliary code is in red). In the optimistic list, a thread traverses the list without taking any locks, and when finding the candidate nodes, it locks the nodes and validates that they are still in the list and adjacent. If the validation fails, the nodes are unlocked and the operation is restarted. The code we show here is an optimized version of the code we discussed in Sec. 7. In Fig. 79, the thread locks only one node p before validation. It does not lock c until the thread is going to remove c at line 35. We prove it is deadlock-free.

Fig. 80 defines the precise invariant and the precondition. Fig. 81 defines the rely/guarantee conditions and the definite actions. Other figures in this section give the proof outlines.

```
intSet gn; //initially empty

struct Node {
  int lock;
  int data;
  struct Node *next;
}
```

```
struct List {
  struct Node *Head;
}

initialize(){
  Head := cons(0, MIN, null);
  Head.next := cons(0, MAX, null);
}
```

```
add(e) {
 1  local p, c, x, s, done, b, w, v, u, r;
 2  done := false;
 3  while (!done) {
 4    p := Head;
 5    c := p.next;
 6    u := c.data;
 7    while (u < e) {
 8      p := c;
 9      c := c.next;
10      u := c.data;
11    }
12    b := false;
13    while (!b) {
14      b := cas(p.lock, 0, cid);
15    }
16    v := p.data;
17    s := Head;
18    w := s.data;
19    while (w < v) {
20      s := s.next;
21      w := s.data;
22    }
23    if (s = p && p.next = c) {
24      done := true;
25    } else {
26      p.lock := 0;
27    }
28  }
29  if (u != e) {
30    x := cons(0, e, c);
31    p.next := x;
32    r := true;
33  } else {
34    r := false;
35  }
36  p.lock := 0;
37  return r;
}
```

```
rmv(e) {
 1  local p, c, n, s, done, b, w, v, u, r;
 2  done := false;
 3  while (!done) {
 4    p := Head;
 5    c := p.next;
 6    u := c.data;
 7    while (u < e) {
 8      p := c;
 9      c := c.next;
10      u := c.data;
11    }
12    b := false;
13    while (!b) {
14      b := cas(p.lock, 0, cid);
15    }
16    v := p.data;
17    s := Head;
18    w := s.data;
19    while (w < v) {
20      s := s.next;
21      w := s.data;
22    }
23    if (s = p && p.next = c) {
24      done := true;
25    } else {
26      p.lock := 0;
27    }
28  }
29  if (u = e) {
30    b := false;
31    while (!b) {
32      b := cas(c.lock, 0, cid);
33    }
34    n := c.next;
35    <p.next := n; gn := gn ∪ {c}>;
36    c.lock := 0;
37    r := true;
38  } else {
39    r := false;
40  }
41  p.lock := 0;
42  return r;
}
```

Figure 79. Optimistic list with TAS lock.

$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1 \qquad \mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$

$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{x \in \mathtt{gn}} \mathsf{N}_{\_}(x, \_, \_)$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N}_s(x, v, z) * \mathsf{ls}_{L'}(z, A', y))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', s, L'.\ A = v :: A' \wedge L = s :: L' \wedge \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y))$

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

$\mathsf{N}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L}_s(p, v, y) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \wedge \mathsf{L}_s(p, v, y)$

$\mathsf{U}(p, v, y) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y)$

$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \wedge (\mathsf{t} \neq s)$

$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A)$

$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = \mathtt{MIN} :: A' :: \mathtt{MAX}) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} = A')$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases}$$

Figure 80. Invariant and precondition of the optimistic list with TAS lock.

As for the lock-coupling list, the invariant I defined in Figure 80 requires the concrete list to be sorted and its elements to constitute the abstract set S. Since the optimistic algorithm ignores the locks when traversing the list, it may access nodes that have been removed from the list. Thus we cannot dispose removed nodes as in the lock-coupling list. Instead, we need to introduce a write-only auxiliary variable gn to remember those removed nodes (see line 35 in Fig. 79). The precise invariant I should include those nodes (garb).

As shown in Figure 81, the delaying actions of a thread t are stratified. The *Add* and *Rmv* actions which update the list are classified at level 2. The lock acquirements *Lock* are at level 1. The *Unlock* actions are not delaying actions.

The definite actions $\mathcal{D}$ describe the various scenarios under which the lock release would definitely happen. Note that it also includes the lock release actions on removed nodes.

The add method is given $\blacklozenge(1, 1)$ initially, where the 2-level $\blacklozenge$-token is for doing *Add* and the 1-level $\blacklozenge$-token is for doing *Lock*. Fig. 82 and Fig. 83 show the proof outlines. Note that when the environment threads perform *Add* or *Rmv*, the current thread could gain more 1-level $\blacklozenge$-tokens (by the stability checking) so that it can rollback and re-do its *Lock* action. In the proofs, the assertions for inner loops are in cyan color, to be distinguished from the assertions for the outer loop. As we mentioned, we apply the (HIDE-$\Diamond$) rule to reuse the proofs of the inner loop at the outer loop, which allows us to discard the tokens of the inner loop. Following earlier work [23], we also provide the (FR-CONJ) rule to add back the original tokens of the outer loop.

The rmv method is given $\blacklozenge(1, 2)$ initially, where the 2-level $\blacklozenge$-token is for doing *Rmv* and the two 1-level $\blacklozenge$-tokens are for doing the two *Lock* actions respectively. The proof in Fig. 84 is similar to the proof of the add method.

$R_{\mathsf{t}} \overset{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$

$G_{\mathsf{t}} \overset{\text{def}}{=} (\mathit{Add}_{\mathsf{t}} \vee \mathit{Rmv}_{\mathsf{t}} \vee \mathit{Lock}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$

$\mathit{Add}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, z, n, v, w, u, s, S.\ ((\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathtt{S} = S)) \ltimes_2 (\mathsf{L}_{\mathsf{t}}(x, v, y) * \mathsf{U}(y, w, z) * (\mathtt{S} = S \cup \{w\})))$
$\quad * [\mathsf{N\_irr}_{\mathsf{t},s}(z, u, n) \wedge (v < w < u)]$

$\mathit{Rmv}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, z, v, u, S, S_g.\ (\mathsf{L}_{\mathsf{t}}(x, v, y) * (\mathsf{gn} = S_g) * (\mathtt{S} = S \uplus \{u\}))$
$\quad \ltimes_2 (\mathsf{L}_{\mathsf{t}}(x, v, z) * (\mathsf{gn} = S_g \cup \{y\}) * (\mathtt{S} = S)) * [\mathsf{L}_{\mathsf{t}}(y, u, z) \wedge (u < \mathtt{MAX})]$

$\mathit{Lock}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, v.\ \mathsf{U}(x, v, y) \ltimes_1 \mathsf{L}_{\mathsf{t}}(x, v, y)$

$\mathit{Unlock}_{\mathsf{t}} \overset{\text{def}}{=}$
$\quad \exists x, y, v.\ \mathsf{L}_{\mathsf{t}}(x, v, y) \ltimes \mathsf{U}(x, v, y)$

$\mathcal{D}_{\mathtt{cid}} \overset{\text{def}}{=} (\forall x.\ dp1_{\mathtt{cid}}(x) \rightsquigarrow dq1_{\mathtt{cid}}(x)) \wedge (\forall x.\ dp2_{\mathtt{cid}}(x) \rightsquigarrow dq2_{\mathtt{cid}}(x))$

$dp1_{\mathsf{t}}(x) \overset{\text{def}}{=}$
$\quad \exists y, z, A, v, u, A', L.\ \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y) * (\mathsf{U}(y, u, z) \vee \mathsf{L}_{\mathsf{t}}(y, u, z)) * \mathsf{ls\_unlocked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A') * \mathsf{garb}$

$dq1_{\mathsf{t}}(x) \overset{\text{def}}{=}$
$\quad \exists y, A, v, A', L.\ \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{U}(x, v, y) * \mathsf{ls\_unlocked}(y, A', \mathtt{null}) * \mathsf{ss}(A :: v :: A') * \mathsf{garb}$

$dp2_{\mathsf{t}}(x) \overset{\text{def}}{=}$
$\quad \exists A, L, S_g.\ \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * (\mathsf{gn} = S_g \uplus \{x\}) * (\circledast_{y \in S_g} \mathsf{N}_{\_}(y, \_, \_)) * \mathsf{L}_{\mathsf{t}}(x, \_, \_)$

$dq2_{\mathsf{t}}(x) \overset{\text{def}}{=}$
$\quad \exists A, L, S_g.\ \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * (\mathsf{gn} = S_g \uplus \{x\}) * (\circledast_{y \in S_g} \mathsf{N}_{\_}(y, \_, \_)) * \mathsf{U}(x, \_, \_)$

Figure 81. Rely, guarantee and definite actions of the optimistic list with TAS lock.

$$\begin{aligned}
&\mathsf{findLocked}(x, v, y, u) \overset{\text{def}}{=}\\
&\quad \exists z, A, A', s, L.\ \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathtt{cid}}(x, v, y) * \mathsf{N\_irr}_{\mathtt{cid},s}(y, u, z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A') * \mathsf{garb}
\end{aligned}$$

$$\begin{aligned}
&p_1(v, A_2, s_1) \overset{\text{def}}{=}\\
&\quad \exists z, A_1, L_1, s, L_2.\ \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head}, A_1, \mathtt{p}) * \mathsf{N}_{s_1}(\mathtt{p}, v, \mathtt{c}) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: v :: \mathtt{u} :: A_2) * \mathsf{garb}
\end{aligned}$$

$$\begin{aligned}
&p_1'(v, A_2, s_1) \overset{\text{def}}{=}\\
&\quad \exists x, z, s, L_2.\ \mathsf{N}_{s_1}(\mathtt{p}, v, x) * \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{c}, \mathtt{u}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{true}\\
&\quad \wedge (\mathtt{p} \in \mathsf{gn} \vee \mathtt{c} \in \mathsf{gn} \vee \exists v', A', L'.\ \mathsf{ls\_irr}_{\mathtt{cid},L'}(x, v' :: A', \mathtt{c}) * \mathsf{true})
\end{aligned}$$

$$p_2(v) \overset{\text{def}}{=} \exists A_2.\ p_2(v, A_2) \qquad p_2(v, A_2) \overset{\text{def}}{=} \exists s_1.\ P \wedge p_1(v, A_2, s_1) \wedge (s_1 \neq \mathtt{cid}) \qquad p_3(v) \overset{\text{def}}{=} \exists A_2.\ P' \wedge p_1(v, A_2, \mathtt{cid})$$

$$p_2'(v) \overset{\text{def}}{=} \exists A_2.\ p_2'(v, A_2) \qquad p_2'(v, A_2) \overset{\text{def}}{=} \exists s_1.\ P \wedge p_1'(v, A_2, s_1) \wedge (s_1 \neq \mathtt{cid}) \qquad p_3'(v) \overset{\text{def}}{=} \exists A_2.\ P' \wedge p_1'(v, A_2, \mathtt{cid})$$

$$\begin{aligned}
&p_4(A_2) \overset{\text{def}}{=}\\
&\quad \exists z, A_1, L_1, s, L_2.\ \mathsf{ls}_{L_1}(\mathtt{Head}, A_1, \mathtt{s}) * \mathsf{N}_s(\mathtt{s}, \mathtt{w}, z) * \mathsf{ls}_{L_2}(z, A_2, \mathtt{null}) * \mathsf{ss}(A_1 :: \mathtt{w} :: A_2) * \mathsf{garb}
\end{aligned}$$

$$\begin{aligned}
&p_4'(A_2) \overset{\text{def}}{=}\\
&\quad \exists z, s, L_2.\ \mathsf{N\_irr}_{\mathtt{cid},s}(\mathtt{s}, \mathtt{w}, z) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z, A_2, \mathtt{null}) * \mathsf{true} \wedge (\mathtt{s} \in \mathsf{gn})
\end{aligned}$$

```
add(e) {
  local p, c, x, s, done, b, w, v, u, r;
  $\{ P \wedge \blacklozenge(1,1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
  done := false;
  $\{ (\neg\mathsf{done} \wedge P \wedge \blacklozenge(1,1) \wedge \lozenge(1)\ \vee\ \exists v.\ \mathsf{done} \wedge \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0) \wedge (v < \mathtt{e} \leq \mathtt{u})) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
  while (!done) {
    $\{ \neg\mathsf{done} \wedge P \wedge \blacklozenge(1,1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    p := Head;
    c := p.next;
    u := c.data;
    $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v, A_2) \wedge \blacklozenge(1,1) \vee p_2'(v, A_2) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \wedge (v < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v, A_2) \wedge \blacklozenge(1,1) \vee p_2'(v, A_2) \wedge \blacklozenge(1,2)) \wedge \lozenge(1 + \mathsf{len}(A_2)) \wedge (v < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    while (u < e) {
      $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v, A_2) \wedge \blacklozenge(1,1) \vee p_2'(v, A_2) \wedge \blacklozenge(1,2)) \wedge \lozenge(\mathsf{len}(A_2)) \wedge (\mathtt{u} < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
      p := c;
      c := c.next;
      u := c.data;
      $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v, A_2) \wedge \blacklozenge(1,1) \vee p_2'(v, A_2) \wedge \blacklozenge(1,2)) \wedge \lozenge(1 + \mathsf{len}(A_2)) \wedge (v < \mathtt{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    }
    $\{ \neg\mathsf{done} \wedge \exists v.\ (p_2(v) \wedge \blacklozenge(1,1) \vee p_2'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    b := false;
    $\left\{ \begin{array}{l} \neg\mathsf{done} \wedge \exists v.\ (\mathtt{b} \wedge (p_3(v) \wedge \blacklozenge(1,0) \vee p_3'(v) \wedge \blacklozenge(1,1) \wedge \lozenge(1)) \vee \neg\mathtt{b} \wedge (p_2(v) \wedge \blacklozenge(1,1) \vee p_2'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1))) \\ \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array} \right\}$
    $\left\{ \begin{array}{l} \neg\mathsf{done} \wedge \exists v.\ (\mathtt{b} \wedge (p_3(v) \wedge \blacklozenge(1,0) \vee p_3'(v) \wedge \blacklozenge(1,1)) \vee \neg\mathtt{b} \wedge (p_2(v) \wedge \blacklozenge(1,1) \vee p_2'(v) \wedge \blacklozenge(1,2)) \wedge \lozenge(1)) \\ \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \end{array} \right\}$
    while (!b) {
      b := cas(p.lock, 0, cid);
    }
    $\{ \neg\mathsf{done} \wedge \exists v.\ (p_3(v) \wedge \blacklozenge(1,0) \vee p_3'(v) \wedge \blacklozenge(1,1) \wedge \lozenge(1)) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    v := p.data;
    s := Head;
    w := s.data;
    $\{ \neg\mathsf{done} \wedge \exists A_2.\ (p_4(A_2) \vee p_4'(A_2)) \wedge (p_3(\mathtt{v}) \wedge \blacklozenge(1,0) \vee p_3'(\mathtt{v}) \wedge \blacklozenge(1,1) \wedge \lozenge(1)) \wedge (\mathtt{v} < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    $\{ \neg\mathsf{done} \wedge \exists A_2.\ (p_4(A_2) \vee p_4'(A_2)) \wedge \lozenge(1 + \mathsf{len}(A_2)) \wedge (p_3(\mathtt{v}) \wedge \blacklozenge(1,0) \vee p_3'(\mathtt{v}) \wedge \blacklozenge(1,1)) \wedge (\mathtt{v} < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    while (w < v) {
      s := s.next;
      w := s.data;
    }
    $\{ \neg\mathsf{done} \wedge \exists A_2.\ (p_4(A_2) \vee p_4'(A_2)) \wedge (p_3(\mathtt{v}) \wedge \blacklozenge(1,0) \vee p_3'(\mathtt{v}) \wedge \blacklozenge(1,1) \wedge \lozenge(1)) \wedge (\mathtt{v} \leq \mathtt{w}) \wedge (\mathtt{v} < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    if (s = p && p.next = c) {
      $\{ \neg\mathsf{done} \wedge p_3(\mathtt{v}) \wedge \blacklozenge(1,0) \wedge (\mathtt{v} < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
      done := true;
      $\{ \mathsf{done} \wedge \mathsf{findLocked}(\mathtt{p}, \mathtt{v}, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0)) \wedge (\mathtt{v} < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    } else {
      p.lock := 0;
      $\{ \neg\mathsf{done} \wedge P \wedge \blacklozenge(1,1) \wedge \lozenge(1) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E}) \}$
    }
  }
  $\{ \exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E}) \}$
  ...
```

Figure 82. Proof outline for add (1).

$p_5 \stackrel{\text{def}}{=} \exists v, x, e.\ \mathsf{findLocked}(\mathtt{p}, v, x, e)$

```
  $\{\exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0) \wedge (v < \mathtt{e} \leq \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})\}$
  if (u != e) {
    $\{\exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})\}$
    x := cons(0, e, c);
    p.next := x;
    r := true;
    $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  } else {
    $\{\exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,0) \wedge (\mathtt{e} = \mathtt{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{e} = \mathtt{E})\}$
    r := false;
    $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  }
  $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  p.lock := 0;
  $\{P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  return r;
}
```

Figure 83. Proof outline for `add` (2).

$\mathsf{adjLocked}(x, v, y, u, z) \stackrel{\text{def}}{=}$
$\quad \exists A, A', L.\ \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathtt{cid}}(x, v, y) * \mathsf{L}_{\mathtt{cid}}(y, u, z) * \mathsf{ls\_irr}_{\mathtt{cid},L}(z, A', \mathtt{null}) * \mathsf{ss}(A :: v :: u :: A') * \mathsf{garb}$

```
rmv(e) {
  local p, c, n, s, done, b, v, u, r;
  $\{P \wedge \blacklozenge(1,2) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{MIN} < \mathtt{e} < \mathtt{MAX}) \wedge (\mathtt{e} = \mathtt{E})\}$
  done := false;
  while (!done) {
    ...
  }
  $\{\exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,1) \wedge (v < \mathtt{e} \leq \mathtt{u} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E})\}$
  if (u = e) {
    $\{\mathsf{findLocked}(\mathtt{p}, \_, \mathtt{c}, \mathtt{e}) \wedge \blacklozenge(1,1) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E} < \mathtt{MAX})\}$
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    $\{\exists z.\ \mathsf{adjLocked}(\mathtt{p}, \_, \mathtt{c}, \mathtt{e}, z) \wedge \blacklozenge(1,0) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E} < \mathtt{MAX})\}$
    n := c.next;
    $\{\mathsf{adjLocked}(\mathtt{p}, \_, \mathtt{c}, \mathtt{e}, \mathtt{n}) \wedge \blacklozenge(1,0) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E} < \mathtt{MAX})\}$
    <p.next := n;  gn := gn ∪ {c}>;
    $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{R} = \mathsf{true})\}$
    c.lock := 0;
    r := true;
    $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  } else {
    $\{\exists v.\ \mathsf{findLocked}(\mathtt{p}, v, \mathtt{c}, \mathtt{u}) \wedge \blacklozenge(1,1) \wedge (v < \mathtt{e} < \mathtt{u}) \wedge \mathsf{arem}(\mathtt{RMV}) \wedge (\mathtt{e} = \mathtt{E})\}$
    r := false;
    $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  }
  $\{p_5 \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  p.lock := 0;
  $\{P \wedge \mathsf{arem}(\mathbf{skip}) \wedge (\mathtt{r} = \mathtt{R})\}$
  return r;
}
```

Figure 84. Proof outline for `rmv`.

D.5 Lazy list with TAS lock

Fig. 85 shows the code of the lazy list implemented with TAS locks (where the auxiliary code is in red). We prove it is deadlock-free.

Similar to the optimistic list, in lazy list, a thread traverses the list without taking any locks, and when finding the candidate nodes, it locks the nodes and validates that they are still in the list and adjacent. But now every node in the concrete list has an additional `mark` field. The `rmv(e)` method first logically removes the node by setting its `mark` field before the physical removal (unlinking it from the list). Since we mainly focus on progress properties, here we only show the proofs for the `add` and `rmv` methods, which involve blocking operations (i.e., lock acquirements). The `contain` method of the lazy list does not acquire locks. It can be verified using earlier technique for linearizability verification (e.g., [21, 30]), and we omit its proofs here.

Fig. 86 defines the precise invariant and the precondition. Fig. 87 defines the rely/guarantee conditions and the definite actions. Other figures in this section give the proof outlines.

As for the optimistic list, we introduce a write-only auxiliary variable `gn` to remember the removed nodes. We also introduce the auxiliary variable `tmark` for each thread to indicate whether the thread has performed marking but has not perform the physical removal.

```
bool[] tmark; //initially all false

intSet gn; //initially empty

struct Node {
  int lock;
  int data;
  struct Node *next;
  bool mark;
}
```

```
struct List {
  struct Node *Head;
}

initialize(){
  Head := cons(0, MIN, null, false);
  Head.next := cons(0, MAX, null, false);
}
```

```
add(e) {
  local p, c, x, done, b, u, r;
  done := false;
  while (!done) {
    p := Head;
    c := p.next;
    u := c.data;
    while (u < e) {
      p := c;
      c := c.next;
      u := c.data;
    }
    b := false;
    while (!b) {
      b := cas(p.lock, 0, cid);
    }
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    if (!p.mark && !c.mark && p.next = c)
      done := true;
    else {
      p.lock := 0;
      c.lock := 0;
    }
  }
  c.lock := 0;
  if (u != e) {
    x := cons(0, e, c, false);
    p.next := x;
    r := true;
  } else {
    r := false;
  }
  p.lock := 0;
  return r;
}
```

```
rmv(e) {
  local p, c, n, done, b, u, r;
  done := false;
  while (!done) {
    p := Head;
    c := p.next;
    u := c.data;
    while (u < e) {
      p := c;
      c := c.next;
      u := c.data;
    }
    b := false;
    while (!b) {
      b := cas(p.lock, 0, cid);
    }
    b := false;
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    if (!p.mark && !c.mark && p.next = c)
      done := true;
    else {
      p.lock := 0;
      c.lock := 0;
    }
  }
  if (u = e) {
    <c.mark := true; tmark_cid := true>;
    n := c.next;
    <p.next := n; gn := gn ∪ {c}; tmark_cid := false>;
    r := true;
  } else {
    r := false;
  }
  p.lock := 0;
  c.lock := 0;
  return r;
}
```

Figure 85. Lazy list with TAS lock.

$A ::= \epsilon \mid v :: A \qquad s ::= 0 \mid \mathsf{t} \qquad L ::= \epsilon \mid s :: L \qquad M ::= \epsilon \mid b :: M \qquad B \in \mathit{ThrdID} \rightharpoonup \mathit{Bool}$

$|\emptyset| \stackrel{\text{def}}{=} 0 \qquad |S \cup \{x\}| \stackrel{\text{def}}{=} |S| + 1 \qquad \mathsf{len}(\epsilon) \stackrel{\text{def}}{=} 0 \qquad \mathsf{len}(v :: A) \stackrel{\text{def}}{=} \mathsf{len}(A) + 1$

$I \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{tmarks} * \mathsf{ls}_L(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$P_{\mathsf{t}} \stackrel{\text{def}}{=} P_{\mathsf{t}}(\mathsf{false}) \qquad P_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists A, L.\ \mathsf{tmarks}_{\mathsf{t}}(b) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * \mathsf{garb}$

$\mathsf{tmarks} \stackrel{\text{def}}{=} \exists B.\ \mathsf{tmarks}(B) \qquad \mathsf{tmarks}(B) \stackrel{\text{def}}{=} (\circledast_{\mathsf{t}} \mathtt{tmark}_{\mathsf{t}} = B(\mathsf{t}))$

$\mathsf{tmarks}_{\mathsf{t}}(b) \stackrel{\text{def}}{=} \exists B.\ \mathsf{tmark}_{\mathsf{t}}(b, B) \qquad \mathsf{tmarks}_{\mathsf{t}}(b, B) \stackrel{\text{def}}{=} (\mathtt{tmark}_{\mathsf{t}} = b) * (\circledast_{\mathsf{t}' \neq \mathsf{t}} \mathtt{tmark}_{\mathsf{t}'} = B(\mathsf{t}'))$

$\mathsf{garb} \stackrel{\text{def}}{=} \circledast_{x \in \mathtt{gn}} \mathsf{N}_{\_}(x, \_, \_, \_)$

$\mathsf{ls}_L(x, A, y) \stackrel{\text{def}}{=} \exists M.\ \mathsf{ls}_L(x, A, y, M) \qquad \mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y) \stackrel{\text{def}}{=} \exists M.\ \mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y, M)$

$\mathsf{ls}_L(x, A, y, M) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge M = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', s, L', b, M'.\ A = (v, b) :: A' \wedge L = s :: L' \wedge M = b :: M' \wedge \mathsf{N}_s(x, v, z, b) * \mathsf{ls}_{L'}(z, A', y, M'))$

$\mathsf{ls\_irr}_{\mathsf{t},L}(x, A, y, M) \stackrel{\text{def}}{=}$
$\quad (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge M = \epsilon \wedge \mathsf{emp})$
$\quad \vee (x \neq y \wedge \exists z, v, A', s, L', b, M'.\ A = (v, b) :: A' \wedge L = s :: L' \wedge M = b :: M' \wedge \mathsf{N\_irr}_{\mathsf{t},s}(x, v, z, b) * \mathsf{ls\_irr}_{\mathsf{t},L'}(z, A', y, M'))$

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

$\mathsf{N}_s(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{N\_irr}_{\mathsf{t},s}(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{lock\_irr}_{\mathsf{t},s}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{L}_s(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{L\_irr}_{\mathsf{t},s}(p, v, y, b) \stackrel{\text{def}}{=} (s \neq \mathsf{t}) \wedge \mathsf{L}_s(p, v, y, b)$

$\mathsf{U}(p, v, y, b) \stackrel{\text{def}}{=} \mathsf{unlocked}(p) * (p.\mathtt{data} = v) * (p.\mathtt{next} = y) * (p.\mathtt{mark} = b)$

$\mathsf{lock}_s(p) \stackrel{\text{def}}{=} \mathsf{locked}_s(p) \vee (s = 0 \wedge \mathsf{unlocked}(p)) \qquad \mathsf{lock\_irr}_{\mathsf{t},s}(p) \stackrel{\text{def}}{=} \mathsf{lock}_s(p) \wedge (\mathsf{t} \neq s)$

$\mathsf{locked}_{\mathsf{t}}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = \mathsf{t}) \wedge (\mathsf{t} \in \mathtt{TIDS}) \qquad \mathsf{unlocked}(p) \stackrel{\text{def}}{=} (p.\mathtt{lock} = 0)$

$\mathsf{s}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN}, \mathsf{false}) :: A' :: (\mathtt{MAX}, \mathsf{false})) \wedge \mathsf{sorted}(A)$

$\mathsf{ss}(A) \stackrel{\text{def}}{=} \exists A'.\ (A = (\mathtt{MIN}, \mathsf{false}) :: A' :: (\mathtt{MAX}, \mathsf{false})) \wedge \mathsf{sorted}(A) \wedge (\mathtt{S} = \mathsf{elems}(A'))$

$$\mathsf{sorted}(A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } A = \epsilon \vee A = (v, b) :: \epsilon \\ (v_1 < v_2) \wedge \mathsf{sorted}((v_2, b_2) :: A') & \text{if } A = (v_1, b_1) :: (v_2, b_2) :: A' \end{cases}$$

$$\mathsf{elems}(A) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } A = \epsilon \\ \{v\} \cup \mathsf{elems}(A') & \text{if } A = (v, \mathsf{true}) :: A' \\ \mathsf{elems}(A') & \text{if } A = (v, \mathsf{false}) :: A' \end{cases}$$

Figure 86. Invariant and precondition of the lazy list with TAS lock.

As shown in Figure 87, the delaying actions of a thread t are stratified. The *Add*, *Rmv* and *Mark* actions which update the list are classified at level 2. The lock acquiring action *Lock11* with certain constraints is at level 1. When a thread does *Lock11*, it must locks a node x which is really on the list and other nodes behind x (i.e., the nodes which are closer to the tail of the list) must be neither locked nor marked. *Lock21* acquires the lock of the successor node with similar constraints. It is also at level 1. Other lock acquirements *Lock1* and *Lock2* are at level 0, which we do not viewed as delaying actions.

The add method is given $\blacklozenge(1, 2)$ initially, where the 2-level $\blacklozenge$-token is for doing *Add* and the two 1-level $\blacklozenge$-tokens are for doing *Lock11* and *Lock21*. Fig. 88 and Fig. 89 show the proof outlines. Note that the thread does not need to lose level-1 $\blacklozenge$-tokens when the node it attempts to lock has been marked by an environment thread. It consumes level-1 $\blacklozenge$-tokens for only *Lock11* and *Lock21* actions. Thus for *Lock1* and *Lock2* actions, the thread still keeps the level-1 $\blacklozenge$-tokens so that it can roll back and do *Lock11* and *Lock21* in the future. Similar to the proofs for the optimistic list, the assertions for inner loops here are also in cyan color, to be distinguished from the assertions for the outer loop. We apply the (HIDE-$\Diamond$) rule to reuse the proofs of the inner loop at the outer loop, which allows us to discard the tokens of the inner loop. Following earlier work [23], we also provide the (FR-CONJ) rule to add back the original tokens of the outer loop.

The rmv method is given $\blacklozenge(2, 2)$ initially, where the two 2-level $\blacklozenge$-token are for doing *Mark* and *Rmv*. The proof in Fig. 90 is similar to the proof of the add method.

$$R_{\mathsf{t}} \stackrel{\text{def}}{=} \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$$

$$G_{\mathsf{t}} \stackrel{\text{def}}{=} (\mathit{Add}_{\mathsf{t}} \vee \mathit{Mark}_{\mathsf{t}} \vee \mathit{Rmv}_{\mathsf{t}} \vee \mathit{Lock11}_{\mathsf{t}} \vee \mathit{Lock21}_{\mathsf{t}} \vee \mathit{Lock1}_{\mathsf{t}} \vee \mathit{Lock2}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\begin{aligned}&\mathit{Add}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, z, n, v, w, u, s, S.\ ((\mathsf{L}_{\mathsf{t}}(x, v, z, \mathsf{false}) * (\mathtt{S} = S)) \ltimes_2 (\mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}) * \mathsf{U}(y, w, z, \mathsf{false}) * (\mathtt{S} = S \cup \{w\})))\\ &\quad * [\mathsf{N\_irr}_{\mathsf{t},s}(z, u, n, \mathsf{false}) \wedge (v < w < u)]\end{aligned}$$

$$\begin{aligned}&\mathit{Mark}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, z, v, u, S.\ [\mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false})]\ *\ ((\mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{false}) * (\mathtt{tmark}_{\mathsf{t}} = \mathsf{false}) * (\mathtt{S} = S \uplus \{u\}) \wedge (u < \mathtt{MAX}))\\ &\quad \ltimes_2 (\mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{true}) * (\mathtt{tmark}_{\mathsf{t}} = \mathsf{true}) * (\mathtt{S} = S)))\end{aligned}$$

$$\begin{aligned}&\mathit{Rmv}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, z, v, u, S_g.\ (\mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}) * (\mathtt{tmark}_{\mathsf{t}} = \mathsf{true}) * (\mathsf{gn} = S_g))\\ &\quad \ltimes_2 (\mathsf{L}_{\mathsf{t}}(x, v, z, \mathsf{false}) * (\mathtt{tmark}_{\mathsf{t}} = \mathsf{false}) * (\mathsf{gn} = S_g \cup \{y\}))\ *\ [\mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{true}) \wedge (u < \mathtt{MAX})]\end{aligned}$$

$$\begin{aligned}&\mathit{Lock11}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, A_1, v, A_2.\ [\mathsf{ls\_irr}_{\mathsf{t},L_1}(\mathtt{Head}, A_1, x) \wedge (v < \mathtt{MAX})] * (\mathsf{U}(x, v, y, \mathsf{false}) \ltimes_1 \mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}))\\ &\quad * [\mathsf{ls\_unlocked\_unmarked}(y, A_2, \mathtt{null})]\end{aligned}$$

$$\begin{aligned}&\mathit{Lock1}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, z, z', v, A_2, A'_2, b, b', L_2, s, L'_2.\ (\mathsf{U}(x, v, y, b) \ltimes \mathsf{L}_{\mathsf{t}}(x, v, y, b))\\ &\quad * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(y, A_2, z) * \mathsf{N\_irr}_{\mathsf{t},s}(z, \_, z', b') * \mathsf{ls\_irr}_{\mathsf{t},L'_2}(z', A'_2, \mathtt{null}) \wedge (b = \mathsf{true} \vee b' = \mathsf{true} \vee s \neq 0)]\end{aligned}$$

$$\begin{aligned}&\mathit{Lock21}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, z, A_1, v, u, A_2, L_1, L_2, M_2.\ [\mathsf{ls\_irr}_{\mathsf{t},L_1}(\mathtt{Head}, A_1, x) * \mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}) \wedge (v < \mathtt{MAX})]\ *\ (\mathsf{U}(y, u, z, \mathsf{false}) \ltimes_1 \mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{false}))\\ &\quad * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(z, A_2, \mathtt{null}, M_2) \wedge (\forall b \in M_2.\ b = \mathsf{false})]\end{aligned}$$

$$\begin{aligned}&\mathit{Lock2}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, y', z, v, u, A_2, b_1, b_2, L_2, M_2.\ [\mathsf{L}_{\mathsf{t}}(x, v, y, b_1) \wedge (v < \mathtt{MAX})]\ *\ (\mathsf{U}(y', u, z, b_2) \ltimes \mathsf{L}_{\mathsf{t}}(y', u, z, b_2))\\ &\quad * [\mathsf{ls\_irr}_{\mathsf{t},L_2}(z, A_2, \mathtt{null}, M_2) * \mathsf{true} \wedge (b_1 = \mathsf{true} \vee b_2 = \mathsf{true} \vee \mathsf{true} \in M_2 \vee \exists v', A', L'.\ \mathsf{ls\_irr}_{\mathtt{cid},L'}(y, (v', \_) :: A', y') * \mathsf{true})]\end{aligned}$$

$$\begin{aligned}&\mathit{Unlock}_{\mathsf{t}} \stackrel{\text{def}}{=}\\ &\quad \exists x, y, v, b.\ [\mathsf{tmarks}_{\mathsf{t}}(\mathsf{false})]\ *\ (\mathsf{L}_{\mathsf{t}}(x, v, y, b) \ltimes \mathsf{U}(x, v, y, b))\end{aligned}$$

$$\mathcal{D}_{\mathtt{cid}} \stackrel{\text{def}}{=} (\forall x.\ dp_{\mathtt{cid}}(x) \rightsquigarrow dq_{\mathtt{cid}}(x)) \wedge (\forall x.\ dp'_{\mathtt{cid}}(x) \rightsquigarrow dq'_{\mathtt{cid}}(x)) \wedge (\forall x.\ dp''_{\mathtt{cid}}(x) \rightsquigarrow dq''_{\mathtt{cid}}(x))$$

$$\begin{aligned}&dp_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists y, z, A, v, u, A', L.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}) * (\mathsf{U}(y, u, z, \mathsf{false}) \vee \mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{false}))\\ &\quad * \mathsf{ls\_unlocked\_unmarked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: (u, \mathsf{false}) :: A') * \mathsf{garb}\end{aligned}$$

$$\begin{aligned}&dq_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists y, z, A, v, u, A', L.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{U}(x, v, y, \mathsf{false}) * (\mathsf{U}(y, u, z, \mathsf{false}) \vee \mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{false}))\\ &\quad * \mathsf{ls\_unlocked\_unmarked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: (u, \mathsf{false}) :: A') * \mathsf{garb}\end{aligned}$$

$$\begin{aligned}&dp'_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists A, L, S_g.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * (\mathsf{gn} = S_g \uplus \{x\}) * (\circledast_{a \in S_g} \mathsf{N}_{\_}(a, \_, \_, \_)) * \mathsf{L}_{\mathsf{t}}(x, \_, \_, \_)\end{aligned}$$

$$\begin{aligned}&dq'_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists A, L, S_g.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, \mathtt{null}) * \mathsf{ss}(A) * (\mathsf{gn} = S_g \uplus \{x\}) * (\circledast_{a \in S_g} \mathsf{N}_{\_}(a, \_, \_, \_)) * \mathsf{U}(x, \_, \_, \_)\end{aligned}$$

$$\begin{aligned}&dp''_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists y, z, A, v, u, A', L.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false}) * \mathsf{L}_{\mathsf{t}}(y, u, z, \mathsf{true})\\ &\quad * \mathsf{ls\_unlocked\_unmarked}(z, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: (u, \mathsf{true}) :: A') * \mathsf{garb}\end{aligned}$$

$$\begin{aligned}&dq''_{\mathsf{t}}(x) \stackrel{\text{def}}{=}\\ &\quad \exists y, z, A, v, u, A', L.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{true}) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head}, A, x) * \mathsf{L}_{\mathsf{t}}(x, v, y, \mathsf{false})\\ &\quad * \mathsf{ls\_unlocked\_unmarked}(y, A', \mathtt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: A') * \mathsf{garb}\end{aligned}$$

Figure 87. Rely, guarantee and definite actions of the lazy list with TAS lock.

$$\mathsf{adjLocked}(x,v,y,u,b,b',b_m) \overset{\text{def}}{=}$$
$$\exists z,A,A',L.\ \mathsf{tmarks}(b_m) * \mathsf{ls\_irr}_{\mathtt{cid},L}(\mathtt{Head},A,x) * \mathsf{L}_{\mathtt{cid}}(x,v,y,b) * \mathsf{L}_{\mathtt{cid}}(y,u,z,b') * \mathsf{ls\_irr}_{\mathtt{cid},L}(z,A',\mathtt{null}) * \mathsf{ss}(A :: (v,b) :: (u,b') :: A') * \mathsf{garb}$$

$$p_0(b_m) \overset{\text{def}}{=} \exists A,L,M.\ \mathsf{tmarks}_{\mathsf{t}}(b_m) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head},A,\mathtt{null},M) * \mathsf{ss}(A) * \mathsf{garb} \wedge (\forall b \in M_2.\ b = \mathsf{false})$$

$$p_0'(b_m) \overset{\text{def}}{=} \exists A,L,M.\ \mathsf{tmarks}_{\mathsf{t}}(b_m) * \mathsf{ls\_irr}_{\mathsf{t},L}(\mathtt{Head},A,\mathtt{null},M) * \mathsf{ss}(A) * \mathsf{garb} \wedge (\mathsf{true} \in M_2)$$

$$p_1(v,A_2,s_1,s_2) \overset{\text{def}}{=}$$
$$\exists z,A_1,L_1,L_2,M_2.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_1}(\mathtt{Head},A_1,\mathsf{p}) * \mathsf{N}_{s_1}(\mathsf{p},v,\mathsf{c},\mathsf{false}) * \mathsf{N}_{s_2}(\mathsf{c},\mathsf{u},z,\mathsf{false}) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z,A_2,\mathtt{null},M_2) * \mathsf{true}$$
$$\wedge (\forall b \in M_2.\ b = \mathsf{false})$$

$$p_1'(v,A_2,s_1,s_2) \overset{\text{def}}{=}$$
$$\exists x,z,L_2,b_1,b_2,M_2.\ \mathsf{tmarks}_{\mathsf{t}}(\mathsf{false}) * \mathsf{N}_{s_1}(\mathsf{p},v,x,b_1) * \mathsf{N}_{s_2}(\mathsf{c},\mathsf{u},z,b_2) * \mathsf{ls\_irr}_{\mathtt{cid},L_2}(z,A_2,\mathtt{null},M_2) * \mathsf{true}$$
$$\wedge (b_1 = \mathsf{true} \vee b_2 = \mathsf{true} \vee \mathsf{true} \in M_2 \vee \exists v',A',L'.\ \mathsf{ls\_irr}_{\mathtt{cid},L'}(x,(v',\_) :: A',\mathsf{c}) * \mathsf{true})$$

$p_2(v) \overset{\text{def}}{=} \exists A_2.\ p_2(v,A_2)$ $\qquad$ $p_2(v,A_2) \overset{\text{def}}{=} \exists s_1,s_2.\ P \wedge p_1(v,A_2,s_1,s_2) \wedge (s_1 \neq \mathtt{cid}) \wedge (s_2 \neq \mathtt{cid})$

$p_2'(v) \overset{\text{def}}{=} \exists A_2.\ p_2'(v,A_2)$ $\qquad$ $p_2'(v,A_2) \overset{\text{def}}{=} \exists s_1,s_2.\ P \wedge p_1'(v,A_2,s_1,s_2) \wedge (s_1 \neq \mathtt{cid}) \wedge (s_2 \neq \mathtt{cid})$

$p_3(v) \overset{\text{def}}{=} \exists A_2,s_2.\ P' \wedge p_1(v,A_2,\mathtt{cid},s_2) \wedge (s_2 \neq \mathtt{cid})$ $\qquad$ $p_3'(v) \overset{\text{def}}{=} \exists A_2,s_2.\ P' \wedge p_1'(v,A_2,\mathtt{cid},s_2) \wedge (s_2 \neq \mathtt{cid})$

$p_4(v) \overset{\text{def}}{=} \exists A_2.\ P'' \wedge p_1(\mathtt{cid},\mathtt{cid})$ $\qquad$ $p_4'(v) \overset{\text{def}}{=} \exists A_2.\ P'' \wedge p_1'(\mathtt{cid},\mathtt{cid})$

```
add(e) {
  local p, c, x, done, b, u, r;
  $\{ P \wedge \blacklozenge(1,2) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX}) \wedge (\mathsf{e} = \mathtt{E}) \}$
  done := false;
  $\{ (\neg\mathsf{done} \wedge P \wedge \blacklozenge(1,2) \wedge \lozenge(1) \ \vee\ \exists v.\ \mathsf{done} \wedge \mathsf{adjLocked}(\mathsf{p},v,\mathsf{c},\mathsf{u},\mathsf{false},\mathsf{false},\mathsf{false}) \wedge \blacklozenge(1,0) \wedge v < \mathsf{e} \leq \mathsf{u})$
  $\ \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX}) \wedge (\mathsf{e} = \mathtt{E}) \}$
  while (!done) {
    $\{ \neg\mathsf{done} \wedge (p_0(\mathsf{false}) \vee p_0'(\mathsf{false}) \wedge \lozenge(1)) \wedge \blacklozenge(1,2) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathtt{MIN} < \mathsf{e} < \mathtt{MAX}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    p := Head;
    c := p.next;
    u := c.data;
    $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v,A_2) \vee p_2'(v,A_2) \wedge \lozenge(1)) \wedge \blacklozenge(1,2) \wedge (v < \mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    $\{ \neg\mathsf{done} \wedge \exists v, A_2.\ (p_2(v,A_2) \vee p_2'(v,A_2)) \wedge \blacklozenge(1,2) \wedge \lozenge(1 + \mathsf{len}(A_2)) \wedge (v < \mathsf{e} < \mathtt{MAX}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    while (u < e) {
      p := c;
      c := c.next;
      u := c.data;
    }
    $\{ \neg\mathsf{done} \wedge \exists v.\ (p_2(v) \vee p_2'(v) \wedge \lozenge(1)) \wedge \blacklozenge(1,2) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    b := false;
    $\{ \neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \vee \neg\mathsf{b} \wedge (p_2(v) \wedge \blacklozenge(1,2) \vee p_2'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)))$
    $\ \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    $\{ \neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2)) \vee \neg\mathsf{b} \wedge (p_2(v) \wedge \blacklozenge(1,2) \vee p_2'(v) \wedge \blacklozenge(1,2)) \wedge \lozenge(1))$
    $\ \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    while (!b) {
      b := cas(p.lock, 0, cid);
    }
    $\{ \neg\mathsf{done} \wedge \exists v.\ (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    b := false;
    $\{ \neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \vee \neg\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)))$
    $\ \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    $\{ \neg\mathsf{done} \wedge \exists v.\ (\mathsf{b} \wedge (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2)) \vee \neg\mathsf{b} \wedge (p_3(v) \wedge \blacklozenge(1,1) \vee p_3'(v) \wedge \blacklozenge(1,2)) \wedge \lozenge(1))$
    $\ \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    while (!b) {
      b := cas(c.lock, 0, cid);
    }
    $\{ \neg\mathsf{done} \wedge \exists v.\ (p_4(v) \wedge \blacklozenge(1,0) \vee p_4'(v) \wedge \blacklozenge(1,2) \wedge \lozenge(1)) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    if (!p.mark && !c.mark && p.next = c) {
      done := true;
      $\{ \mathsf{done} \wedge \exists v.\ \mathsf{adjLocked}(\mathsf{p},v,\mathsf{c},\mathsf{u},\mathsf{false},\mathsf{false},\mathsf{false}) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    } else {
      $\{ \neg\mathsf{done} \wedge \exists v.\ (p_4(v) \vee p_4'(v)) \wedge \blacklozenge(1,2) \wedge \lozenge(1) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
      p.lock := 0;
      c.lock := 0;
      $\{ \neg\mathsf{done} \wedge \exists v.\ (p_2(v) \vee p_2'(v)) \wedge \blacklozenge(1,2) \wedge \lozenge(1) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
    }
  }
  $\{ \exists v.\ \mathsf{adjLocked}(\mathsf{p},v,\mathsf{c},\mathsf{u},\mathsf{false},\mathsf{false},\mathsf{false}) \wedge \blacklozenge(1,0) \wedge (v < \mathsf{e} \leq \mathsf{u}) \wedge \mathsf{arem}(\mathtt{ADD}) \wedge (\mathsf{e} = \mathtt{E}) \}$
  ...
```

Figure 88. Proof outline for add (1).

$$p_5(v) \overset{\text{def}}{=}$$
$$\exists z, A, A', s, L.\ \mathsf{tmarks}(\mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L}(\texttt{Head}, A, \texttt{p}) * \mathsf{L}_{\texttt{cid}}(\texttt{p}, v, \texttt{c}, \mathsf{false}) * \mathsf{N\_irr}_{\texttt{cid},s}(\texttt{c}, \texttt{u}, z, \mathsf{false})$$
$$* \ \mathsf{ls\_irr}_{\texttt{cid},L}(z, A', \texttt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) :: (\texttt{u}, \mathsf{false}) :: A') * \mathsf{garb}$$

$$p_6 \overset{\text{def}}{=}$$
$$\exists z, A, v, A', L.\ \mathsf{tmarks}(\mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L}(\texttt{Head}, A, \texttt{p}) * \mathsf{L}_{\texttt{cid}}(\texttt{p}, v, z, \mathsf{false}) * \mathsf{ls\_irr}_{\texttt{cid},L}(z, A', \texttt{null}) * \mathsf{ss}(A :: (v, \mathsf{false}) ::: A') * \mathsf{garb}$$

```
      { ∃v. adjLocked(p, v, c, u, false, false, false) ∧ ⧫(1, 0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
27    c.lock := 0;
      { ∃v. p_5(v) ∧ ⧫(1, 0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
28    if (u != e) {
        { ∃v. p_5(v) ∧ ⧫(1, 0) ∧ (v < e < u) ∧ arem(ADD) ∧ (e = E) }
29      x := cons(0, e, c, false);
30      p.next := x;
31      r := true;
        { p_6 ∧ arem(skip) ∧ (r = R) }
32    } else {
        { ∃v. p_5(v) ∧ ⧫(1, 0) ∧ (e = u) ∧ arem(ADD) ∧ (e = E) }
33      r := false;
        { p_6 ∧ arem(skip) ∧ (r = R) }
34    }
      { p_6 ∧ arem(skip) ∧ (r = R) }
35    p.lock := 0;
      { P ∧ arem(skip) ∧ (r = R) }
36    return r;
   }
```

Figure 89. Proof outline for add (2).

```
   rmv(e) {
 1    local p, c, n, done, b, u, r;
      { P ∧ ⧫(2, 2) ∧ arem(RMV) ∧ (MIN < e < MAX) ∧ (e = E) }
 2    done := false;
 3    while (!done) {
..      ...
26    }
      { ∃v. adjLocked(p, v, c, u, false, false, false) ∧ ⧫(2, 0) ∧ (v < e ≤ u < MAX) ∧ arem(RMV) ∧ (e = E) }
27    if (u = e) {
        { adjLocked(p, _, c, e, false, false, false) ∧ ⧫(2, 0) ∧ arem(RMV) ∧ (e = E < MAX) }
28      <c.mark := true; tmark_cid := true>;
        { adjLocked(p, _, c, e, false, true, true) ∧ ⧫(1, 0) ∧ arem(skip) ∧ (e < MAX) ∧ (R = true) }
29      n := c.next;
30      <p.next := n; gn := gn ∪ {c}; tmark_cid := false>;
31      r := true;
        { p_6 ∧ arem(skip) ∧ (r = R) }
32    } else {
        { ∃v. adjLocked(p, v, c, u, false, false, true) ∧ ⧫(2, 0) ∧ (v < e < u) ∧ arem(RMV) ∧ (e = E) }
33      r := false;
        { p_6 ∧ arem(skip) ∧ (r = R) }
34    }
      { p_6 ∧ arem(skip) ∧ (r = R) }
35    p.lock := 0;
36    c.lock := 0;
      { P ∧ arem(skip) ∧ (r = R) }
37    return r;
   }
```

Figure 90. Proof outline for rmv.